VSB - TECHNICAL UNIVERSITY OF OSTRAVA
FACULTY OF ECONOMICS

DEPARTMENT OF SYSTEMS ENGINEERING

Automatic Identification of Slovak Text Author using Machine-Learning Methods

Automatická identifikace autora textu ve slovenštině pomocí metod strojového učení

Student: Bc. Miloš Švaňa

Supervisor of the Diploma Thesis: Ing. Radek Němec, Ph.D.

OSTRAVA 2019

VŠB - Technical University of Ostrava
Faculty of Economics
Department of System Engineering

Diploma Thesis Assignment

Student: **Bc. Miloš Švaňa**

Study Programme: N6209 Systems Engineering and Informatics

Study Branch: 6209T017 Informatics in Economics

Title: Automatic Identification of Slovak Text Author using Machine-Learning Methods

Automatic Identification of Slovak Text Author using Machine-Learning Methods

The thesis language: English

Description:

1. Introduction
2. Theoretical Foundations of Machine Learning and Natural Language Processing
3. Methodology of the Automatic Author Identification
4. Results of Speech Author Identification
5. Conclusion

Bibliography
List of Abbreviations
Declaration of Utilisation of Results from the Diploma Thesis
List of Annexes
Annexes

References:

AGGARWAL, Ch. C. and Ch. X.ZHAI. *Mining Text Data*. New York: Springer, 2012. ISBN 978-1-4614-3222-7.
HAN, J., M. KAMBER and J.PEI. *Data Mining: Concepts and Techniques*, 3rd ed. San Francisco: Morgan Kaufmann Publishers, 2011. ISBN 978-0-12-381479-1.
RASCHKA, S. *Python Machine Learning*. Birmingham: Packt Publishing, 2015. ISBN 978-1-78355513-0.

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **Ing. Radek Němec, Ph.D.**

Date of issue: 23.11.2018
Date of submission: 26.04.2019

doc. Ing. Jana Hančlová, CSc.
Head of Department

prof. Dr. Ing. Zdeněk Zmeškal
Dean

Declaration of Independent Elaboration of a Diploma Thesis

I hereby confirm that I made the available dissertation independently and without use of others than indicated aids. All passages inferred literally or in general manner from published and not published sources, are marked as such. The work was never submitted to any examining authority in this or any similar form.

Ostrava, dated 24. 4. 2019

Miloš Švaňa

I would like to express my sincere gratitude to my supervisor, Ing. Radek Němec, Ph.D. for providing inspiration, guidance and feedback. I would also like to thank my family and two best friends for supporting me during my university studies.

Contents

1 Introduction

One of the most important sources of information are text documents written in natural human languages. People can process this medium with relative ease. However, in recent years the amount of text data produced has been rapidly growing, thanks to phenomena such as social networks. Computers are needed to process such large quantities of data.

Researchers in fields such as natural language processing or machine learning are developing methods to make computer-aided text processing possible. Nowadays, computers can be "trained" to extract useful information from text documents and even partially understand meaning of words. Advancements in the area of natural language understanding enable applications such as machine translation or virtual assistants (Google Assistant, Siri, Cortana).

One of the most important tasks solved by machine learning and natural language processing is text classification. The goal is to develop a model capable of automatically determining a class label (i.e. category) of a previously unseen text document. Text classification has many applications: sentiment analysis, topic modeling, spam filters or author identification.

Vast majority of classification methods can work only with numerical data. Henceforth, one of the key issues of text classification is transforming text documents into a numerical representation. This process is sometimes called vectorization. A traditional way of solving this problem is using some variant of the bag-of-words method. As explained further, this approach is intuitive but comes with a number of disadvantages. Most notably, it produces high-dimensional sparse vectors and ignores meaning and order of words in the document. Hence, more modern approaches addressing these problems were developed: word-graph and embeddings.

In this thesis we are asking the question, whether these new methods, especially the word-graph approach, are useful for solving classification problems. Do they provide advantages in either accuracy or performance compared to traditional bag-of-words approaches? Can they be further improved?

To answer these questions we perform a series of experiments on a sample problem: we intend to create a model capable of identifying an author of a text document written in Slovak. More specifically, we want to recognize a politician making a speech in Slovak national parliament. This problem is characterized by a morphologically rich language, long documents and large amount of classes. During the experiments we focus on comparing

classification accuracy provided by different vectorization and classification methods. The secondary goal is to analyze the time requirements of the training process.

The thesis is structured as follows: In section 2 we review the relevant literature and discuss problems and methods of machine learning and natural language processing. In both cases we start with the basic definitions, then survey some of the problems solved in both areas of research and finally discuss the methods for solving text classification and vectorization.

We describe the methodology of author identification experiments in section 3. The section discusses the scope of the experiments, implementation tools and some of the most important parts of the source code.

Finally, section 4 covers the discussion on experiment results. Based on these, we also propose some modifications of the word-graph method potentially improving classification accuracy.

2 Theoretical Foundations of Machine Learning and Natural Language Processing

Implementing a mechanism for automatic author identification can be understood through the *knowledge discovery from data* process (KDD) described by Han, Kamber and Pei [9]. This process is a sequence of 7 steps:

1. Data cleaning (removal of noisy data)
2. Data integration (combining multiple data sources)
3. Data selection (selection of data relevant for the problem)
4. Data transformation (data preprocessing)
5. Data mining
6. Evaluation
7. Presentation

Given the specific problem described in introduction, steps 1, 2 can be skipped as the data used originate from a single data source already containing consistent data. The selection step is shortly described in section 3.2.

This chapter further introduces theoretical foundations required to successfully implement the transformation and mining (classification) steps. These prerequisites are divided into two main areas: Machine Learning and Natural Language Processing. Many concepts and techniques are shared and referenced in literature focused on either of the two. Additionally, terms like *Data Mining* and *Data Science* are often used to describe even broader fields of research encompassing both areas.

We start the chapter by defining these terms for further needs and discussing the field of Machine learning. In second part we examine tasks and techniques belonging to the area of Natural Language processing focusing on the use of Machine Learning methods in the context of text analysis.

The evaluation step consists of analyzing experiment results in section 4. In this thesis, we ignore the last step of results presentation as we focus mostly on evaluating multiple transformation (i.e. vectorization) and mining(i.e. classification) methods.

2.1 Machine Learning and Related Fields of Research

Raschka [18] defines *machine learning* as a subfield of artificial intelligence focused on developing self-learning alogithms to gain knowledge from data mainly for the purpose of making predictions. The research in this area examines how tasks such as finding patterns, rules and models describing the data can be performed automatically by computers.

Data mining is defined by Han, Kamber and Pei [9] as a process of discovering interesting patterns and knowledge from large amounts of data. Such description can be considered synonymous to the definition of machine learning. The authors note that there are multiple ways of describing the term. To distinguish between machine learning, some also consider techniques related to data cleaning, integration and transformation or knowledge representation to be a part of the data mining.

Most comprehensive field encompassing both data mining and machine learning, and depending on personal opinion also Natural Language processing, is *data science*. It is often defined as an intersection of mathematics/statistics, computer science and domain/business knowledge. This relationship is illustrated by figure 2.1. The diagram also defines machine learning as a combination of computer science and mathematics, excluding domain knowledge.

We further avoid using the term *Data Science* in the text and for the purpose of this thesis, we consider the term *data mining* synonymous to the term *machine learning*.

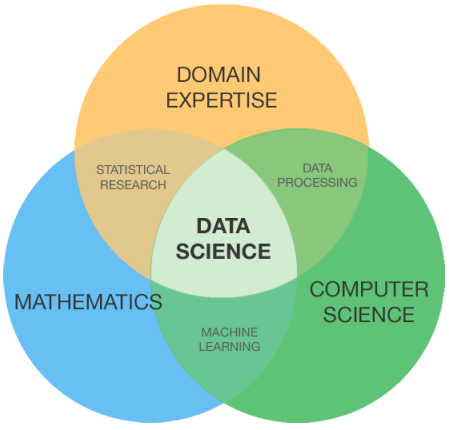

Figure 2.1: Data Science as an interdisciplinary field. Source: [16]

2.1.1 Taxonomy of Machine Learning Methods

Algorithms for solving machine learning problems can be grouped into multiple categories. Following list contains a selection of categories known to the majority of machine learning practioners.

Supervised learning algorithms are often used as a starting point for those who are beginning to study the field of Machine learning. Methods in this group require a set of training data to create a model, that can be later used to make predictions about unseen data samples. Expected output is known in advance for each object in the training set. [18] A supervised learning algorithm then attempts to find a function mapping training set objects to these expected output values. For example sentiment classification model can be created from a set of text documents already marked with "positive impression" or "negative impression" labels.

On the other hand *unsupervised learning* methods are used when there is no desired output known in advance. These approaches are capable of examining the structure of data and identifying previously unknown patterns [18]. Clustering, i.e. separating data into meaningful groups, is a typical unsupervised learning problem.

Reinforcement learning can be described as "reward and punishment" method. Trained model usualy makes a choice to perform certain action and its environment provides feedback in form of a reward function such as current game score. In recent year this area caught the interest of many researchers [18]. New models capable of playing complicated computer games show promising results and in some cases are already able to defeat experienced human opponents.

Multiple definitions are provided to describe the field of *deep learning* [6]. These models are usually built as a cascade of multiple non-linear layers in which an output from one layer is used as an input for the next one. Such layers are then used to learn data representations on multiple levels of abstraction. Areas of application include computer vision, speech recognition or automatic translation. Deep learning models, especially during training phase, require high amounts of computing power highly exceeding capacities of standard CPUs installed on a single device. Instead, clusters of GPUs, FPGAs or specialized Deep learning hardware are often used to execute such algorithms.

2.1.2 Survey of Problems Solved by Machine Learning Methods

This section explores some of the most common problems solved by methods of machine learning. Following topics are discussed:

- Classification
- Regression
- Frequent pattern and association mining
- Outlier detection
- Clustering
- Dimensionality reduction

Description of each problem type includes a short definition, examples of application and a brief survey of methods that can be used to solve the problem.

Since classification is the focus point of this thesis, this section introduces it only very shortly to provide context for other types of problems. Detailed description of classification methods can be found in section 2.1.3.

2.1.2.1 Classification. Classification is a typical supervised learning problem. A set of feature vectors characterizing entities of a certain type, such that each is labeled by a class label from a finite set, is usually provided as training data. The goal of classification is to use this training set to train a model capable of predicting class labels for previously unseen samples [9].

Application possibilities of classification methods are very broad. When it comes to text analysis, classification techniques can be used to filter spam, identify a language an author, automatically tag news articles, or differentiate between positive and negative comments on social networks. When using other types of media as input, even more advanced applications can be realized: recognizing characters and health problems from images, identifying sex or even specific people from a voice recording etc.

Additionally, as mentioned in following subsections, some other types of machine learning problems like outlier detection, can be transformed into a classification problem.

Previous description of classification is sufficient to provide context and to explain analogies with other machine learning problems. Survey of classification methods with focus on text classification can be found in section 2.1.3.

2.1.2.2 Regression. Regression shares many commonalities with classification. In both cases, the goal is to use a set of training data to train a model capable of predicting a value, providing new information about a unseen data sample. Key difference between the two concepts is that in case of classification we are predicting a categorical class label while when performing regression a continuous value is predicted [18].

Alternatively, we can look at training a regression model as at a process of finding a definition of a simple

$$y = f(\vec{x}), \tag{2.1}$$

function, mapping a vector of features $\vec{x}$ to a continuous value output y.

Various applications of regression can be found in the area of finance and economics. Regression models are used to predict stock and commodity price or currency exchange rates. Additionally, regression can be used also for understanding relationships between variables or for analyzing trends [18].

Because of the similarities between classification and regression, many methods used for classification, like Support Vector Machines or Neural Networks, can be also applied to solve regression problems. As already mentioned, some of these methods are examined in section 2.1.3.

2.1.2.3 Frequent pattern and association mining. According to Han, Kamber and Pei [9] *frequent pattern mining* can be defined as searching "for recurring relationships in a given data set". 3 basic types of frequent patterns are distinguished:

- *subsets*, like items often bought together;
- *sequential patterns*, representing cases where order of items is important;
- *structural patterns*, such as frequent sub-graphs or sub-trees.

Han, Kamber and Pei [9] further says that frequent pattern mining is important for discovering associations, correlations or other types of relationships in a dataset.

Typical application of association and correlation mining is customer behavior analysis, allowing businesses to optimize location of goods in physical stores, or to provide recommendations what the customer should buy/do/watch next. Similar features are also offered by many multimedia content providers like YouTube, Netflix or Spotify.

Frequent patterns are often represented in form of associations. For example $iPhone$ =>

iPhone case association can be used to describe the information that customers purchasing iPhones also buy iPhone cases.

Two metrics are often used to determine how interesting an association is:

- *Support* can be defined as the ratio of data set objects behaving according to the association. For example 5% support for the rule above means that 5% of all purchases made by customers contained both an iPhone and an iPhone case.
- *Confidence* is similar to conditional probability. Given that a customer is buying an iPhone, it expresses the probability of also buying a case.

To identify interesting patterns, minimum support and confidence thresholds are usually defined. A business owner, for example, can determine that an association rule is interesting if its support is greater than 3% and its confidence is greater than 40%.

Apriori algorithm can be used to automatically discover frequent patterns in a dataset. Originally proposed by Agrawal and Srikant [2], this method scans the dataset and identifies combinations of k transaction items (starting at $k = 1$), satisfying minimum support threshold and uses the results to recursively identify groups of $k + 1$ items. To further reduce the search space, the algorithm uses the *apriori* property based on the idea that if support of a set of items is lower then minimal support threshold and a new item is added, resulting itemset's support will also not fulfill the threshold requirement.

2.1.2.4 Clustering. Han, Kamber and Pei [9] describes clustering as "the process of partitioning a set of data objects (or observations) into subsets". From a certain point of view the concept of clustering shares similarities with classification. In both cases, we are talking about distinguishing between multiple subsets of data. The main difference between the two is that in case of classification, we have certain prior knowledge about these subsets. We know what subsets (classes) are contained in the set of training data and objects in this training set are already labeled with a class label.

No such information is provided in case of clustering; it is an unsupervised learning problem. The task of clustering algorithms is not to train a model capable of predicting a class label of an unknown object, but rather discover what classes are there in the first place. In other words, clustering methods try to create groups of similar objects. These groups or clusters can often lead to discovery of interesting and previously unknown patterns.

One of the applications of clustering methods is mentioned in the following subsection:

outlier detection. Other uses include cases such as discovering customer groups each sharing certain shopping behavior patterns, or grouping articles into topics (without having any topics defined in advance).

Raschka [18] diferentiates between 3 basic types of clustering algorithms:

- *prototype-based* methods, such as *k*-means represent clusters by prototypes, like centroids (average data points) or medoids (most frequent data points);
- *hierachical* clustering leads to creation of cluster hierarchies - smaller clusters being grouped into larger and larger ones;
- *density-based* methods use density metrics, such as number of data points found in a certain radius, to discover clusters.

K-means is one of the simplest clustering methods. Raschka [18] summarizes it in 4 steps:

1. Randomly pick k data points to serve as initial centroids to represent cluster centers.
2. Assign each object to the nearest centroid.
3. Move the cluster centroids to the center of the objects assigned to it.
4. Repeat steps 2, 3 until maximum number of iterations is reached or the clusters no longer change (i.e. the cluster centers do not move or move within a certain threshold).

Although intuitive, this method has multiple drawbacks, most significant of them being that the number of clusters (k) has to be provided in advance. Additionally, *k*-means deals well with datasets forming spherical structures, but other methods might provide better results in case of other shapes.

2.1.2.5 Outlier Detection. Outlier detection (also called anomaly detection) is a process of finding data objects whose behaviour differs greatly from expectation. Answering *Is this weird?* questions is useful in many areas of human life: fraud detection, medical care, sensor/video network survailence or damage detection. For example, an unusual purchase amount, or location of a payment might be relevant indicators that a credit card was stolen. Additionaly, other Machine learning techniques might be sensitive to outliers [18] and their removal can be recommended as a part of data preprocessing.

Detection of outliers is closely related to the previously discussed clustering problem. Clustering detects how majority of the data behaves and then organizes the data set according

to the recognized patterns. Outlier detection then identifies objects which do not match these majority patterns [9].

Han, Kamber and Pei [9] further defines 3 types of outliers. *Global outliers* differ from all other data points in a dataset. *Context outliers* represent a behavior not expected in certain context. For example temperature of 28 °C is considered normal during summer in central Europe. During winter, it would be an anomaly. Lastly, *collective outliers* only deviate from standard behavior as a group; individual cases are not considered abnormal. Authors provide an example of a delayed order shipment. Delay of one package might not be abnormal, but multiple orders delayed in a short period of time can be classified as an anomaly.

Han, Kamber and Pei [9] lists 4 main challenges when dealing with outlier detection problems:

- modeling normal data;
- specifics of application area;
- handling noise;
- understandability of the results.

Multiple types of methods can be used for outlier detection. Outliers and normal data can be manually labeled by experts in a given area of application and then used to train a classifier, transforming the outlier detection into a classification problem. Alternatively, an unsupervised approach can be taken. Assuming that the normal data points are clustered together and outliers are positioned far from other objects, clustering methods can be applied to detect anomalies [9].

In addition to distinguishing between supervised and unsupervised approaches, Han, Kamber and Pei [9] recognizes multiple categories based on types of assumptions made about the examined set of data:

- *Statistical methods* assume that normal data are generated by a statistical (Gaussian) model.
- *Proximity based methods* assume longer distances between outliers and their closest neighbors compared to normal data.
- *Clustering-Based methods* expect the normal data to group into a large dense cluster and outliers to form small or sparse clusters.

2.1.2.6 Dimensionality Reduction. Many real world phenomena are characterized by high-dimensional feature vectors (high number of attributes). Dimensionality reduction techniques aim to reduce the number of feature vector dimensions with minimal information loss.

Raschka [18] sums up two main reasons for using dimensionality reduction techniques:

1. Dimensionality reduction can be used to reduce overfitting. Overfitting happens when model parameters are fitted too closely to the observations in the training set resulting into weak generalization to unseen data.
2. It allows researchers and businesses to effectively store and analyze ever increasing amounts of data.

As explained in following sections, dimensionality reduction is very important part of the text classification process. Application of many text vectorization techniques leads to creation of high-dimensional feature spaces, usually copying the vocabulary size (number of unique tokens or words in the dataset), resulting into feature vectors with hundreds of thousands of dimensions.

Two main approaches are used to solve the dimensionality reduction task: *feature selection* and *feature extraction*.

Feature selection aims at selecting a subset of k existing features best describing given problem. One option how to select best performing subset of features is using *sequential backward selection* algorithm. This method removes features from the original set until only desired number of features is left. Most important part of the process is choosing correct features to remove. Backward selection uses criterion function minimization. Raschka [18] suggests that a criterion function can be defined simply as the difference between the performance of the algorithm before and after feature removal.

On the other hand, feature extraction creates a new, smaller set of features using original feature vectors. 3 feature extraction techniques are mentioned by Raschka [18]:

- unsupervised *Principal component analysis (PCA)*,
- supervised *Linear Discriminant Analysis (LDA)*,
- *kernel principal component analysis*.

To provide an example, the PCA method is used to reduce feature vector complexity by transforming data into a lower-dimensional space. These new features then summarize

the information contained in the original feature set. The transformation is realized by geometrically projecting data points onto special dimensions called principal components. Principal components are selected in a way that minimizes the distance between data points and their projections and maximizes variance at the same time. Additionally, principal components should be uncorrelated (orthogonal) [13].

2.1.3 Classification Problem and Classification Methods

Classification represents the data mining step of the knowledge discovery process described in the beginning of chapter 2. As mentioned in previous section, classification can be defined as a creation of a model capable of predicting a class label (category a certain data point belongs to) from a finite set known in advance. To create such a model, a training data set is required. This set contains vectors of features describing certain objects or measurement records. Important property of the training set is that each object has a class label already assigned. For this reason classification belongs to the group of supervised learning methods [1].

The problem presented in this thesis ultimately leads to text data classification. Therefore, this section examines this category of machine learning tasks in depth. In subsection 2.1.3.1 we introduce basic subtypes of classification problems. Rest of this section then describes different methods used for solving classification problems recommended by Aggarwal and Zhai [1] for text classification. It is important to note that all of these methods can be applied to any type of data (as long as it can be transformed into numerical representation).

2.1.3.1 Basic Types of Classifiers.

Depending on the model results, we can distinguish between *hard* and *soft* classification. For a tested sample, hard classification simply returns the class label predicted by the model. On the other hand, soft classification provides information about probability with which the sample belongs to each of the classes. One of the options how to create a soft classifier is using logistic sigmoid function defined as:

$$\phi(z) = \frac{1}{1 + e^{-z}} \tag{2.2}$$

As illustrated by figure 2.2 this function maps an input from interval $(-\infty; \infty)$ to an output from a probability modeling interval $[0; 1]$. Raschka [18] further says that this function is an inverse of the logit (log-odds) function which expects probability as input.

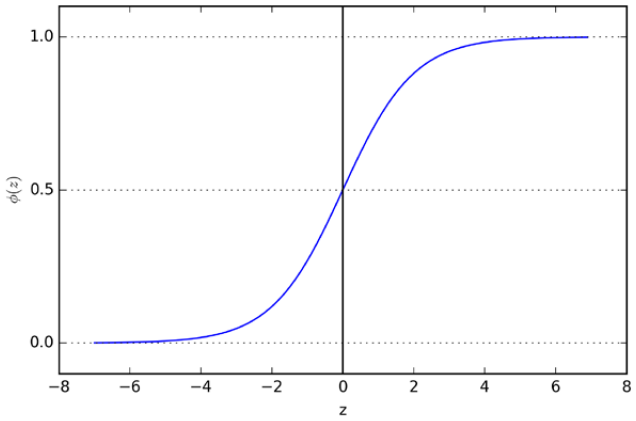

Figure 2.2: Logistic sigmoid function. Source: Raschka [18]

Some of the classification methods, like Support Vector Machines, are capable of solving only *binary* classification problems, i.e. distinguishing between two classes (usually labeled as 0 and 1 or -1 and 1). In support vector machines review, we introduce some techniques that can be used to extend the capabilities of these classifiers to cover also the *multi-class* problems where the task is to choose from more then two class labels. Other methods, such as Neural Networks, are able to solve tasks with more than 2 classes without any modifications. Finally, there is a category of *multi-label* classification problems characterized by the fact that multiple labels can be assigned to one sample. This is not the case of the author identification problem, therefore this type will not be explained in following text.

2.1.3.2 Decision Trees and Random Forests. According to Aggarwal and Zhai [1], decision trees are used to hierarchically decompose the data space using predicates applied on the feature values. Raschka [18] compares decision trees to a series of simple questions that are used to divide the data. This division of data leads to creation of rectangular borders between classes as can be observed in figure 2.3.

When predicting a class label for an unknown sample, the algorithm tests values in the feature vector representing the sample against the predicate in the root node of the tree. Depending on the result of the test, it then moves down to a corresponding child node. The process is repeated until a leaf node is reached. This leaf node then represents a final class label. Figure 2.4 illustrates a simple decision tree that can be used for answering the question: *What should I do today?*

Raschka [18] emphasises that one of the biggest benefits of decision tree classifiers is its iterpretability; it is easy to understand why a particular class label was chosen for given

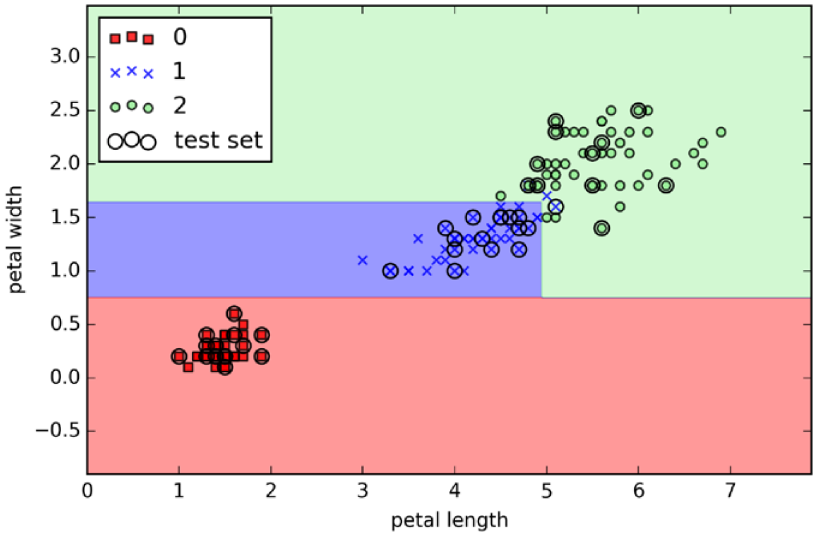

Figure 2.3: Class borders created by a decision tree. Source: Raschka [18]

sample. This might not be true for other classification methods especially for complex neural networks with multiple layers. Han, Kamber and Pei [9] further expands the list of benefits of using decision trees for classification:

- decision trees do not require extensive domain knowledge,
- very low number of parameters has to be set when training decision trees (usually only maximum depth),
- they provide intiutive knowledge representation,
- fast and simple process of training and classification.

Classification is not the only field of application of decision trees. As mentioned in previous sections, many classification methods can be used for regression. Popular Python library Scikit-learn provides a *DecisionTreeRegressor* class[1] allowing users to use decision trees for this purpose as well. Another use of decision trees is feature selection (see section 2.1.2.6). During decision-tree construction, feature importance can be calculated and the results can be used to select most important features to be utilized by other ML methods [18].

The process of decision tree-construction is called *decision tree induction* [9]. In most cases, training a classifier or a regressor leads to solving one or more optimization problems. To construct a decision tree at each node, an *information gain* maximizing feature is chosen

[1]https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

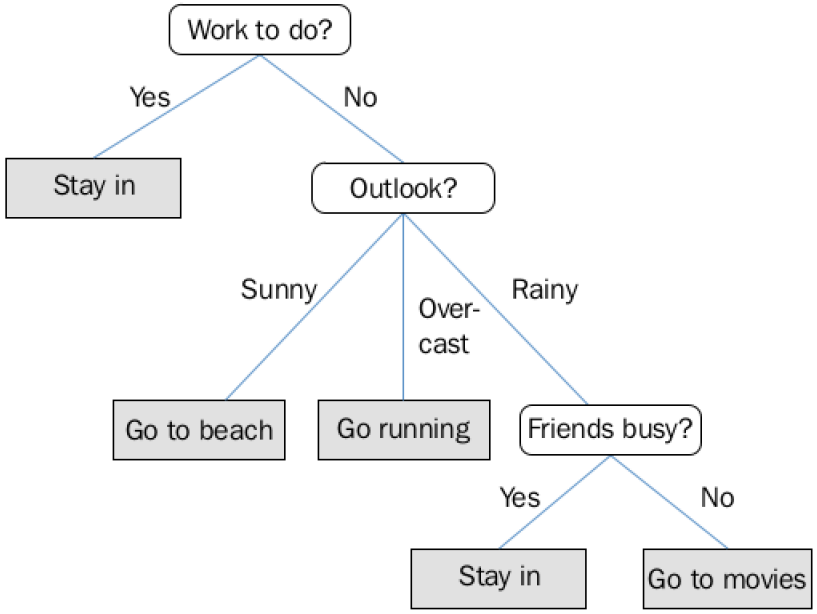

Figure 2.4: Simple *What should I do today?* decision tree. Source: Raschka [18]

[18]. This metric is defined as follows:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j) \tag{2.3}$$

In equation 2.3:

- f is a feature used for splitting the dataset into smaller parts,
- D_p is the set of data objects belonging to the parent node,
- D_j is the set of data objects belonging to j-th child,
- N_p and N_j is the number of data objects belonging to the parent and j-th child node respectively,
- and I is an impurity measure.

Raschka [18] talks about 3 impurity measures commonly used. *Entropy* is defined as:

$$E(D) = \sum_{i=1}^{c} -p_i \; log_2 \; p_i \tag{2.4}$$

where p_i is the ratio of data objects belonging to i-th class from a set of all classes in the dataset c [23]. Entropy is maximized if classes are distributed evenly (in case of a binary problem if 50% of objects belong to class 0 and the other 50% to class 1). On the other hand, 0 entropy is reached only if objects from one class are present in the node.

Other common impurity measure is *Gini index*:

$$G(D) = 1 - \sum_{i=1}^{c} p_i^2 \tag{2.5}$$

Gini index behaves similarly to entropy and according to Raschka [18], both measures provide very similar results in terms of classification performance. Last measure mentioned is classification error calculated from the perspective of node's most common class:

$$I_E = 1 - max(p_i) \tag{2.6}$$

Multiple decision trees can be combined into *random forests* - one of ensemble learning techniques. In ensemble learning, multiple classifiers are grouped together to provide better accuracy as would be possible for any of the classifiers alone. When predicting a class label, each classifier in the group provides it's own result and then a voting mechanism, such as choosing most common answer, is then used to determine the final output [18].

When constructing a tree belonging to a random forest of size k, the algorithm selects n random samples (with replacement) from the dataset and uses only d randomly selected features (without replacement).

Random forests usually provide better classification accuracy than a single decision tree. The cost of this improvement is then lower interpretability of the final model [18].

2.1.3.3 Support Vector Machine. Support Vector Machine (in its basic form) is a linear classification method. Linear classifiers use hyperplanes (for example a line in a 2-dimensional space) to separate individual classes instead of more complex border shapes created for example by decision trees. These hyperplanes are defined by a vector of weights $\vec{w}$. In a simple example of a line defined by equation $y = ax + b$, the vector of weights can be either $\vec{w} = (a)$ or $\vec{w} = (b, a)$. The latter also containing the bias b is often called augmented. Both variants are used in practice. If the algorithm uses the augmented vector, it usually also augments the feature vectors by prepending the value 1 in front. To predict a class label after a linear classifier is trained, a simple hypothesis function is used [2]:

$$y(\vec{x}) = signum(\vec{w} \cdot \vec{x}) \tag{2.7}$$

In equation 2.7, $\vec{w}$ is an augmented vector of weights representing a hyperplane and $\vec{x}$ is an augmented feature vector. Since the signum function can return only one of two values (1 or -1) linear classifiers using this hypothesis function can only solve binary classification problems.

When training a linear classifier, the goal is then to find vector $\vec{w}$. Many linear classifiers, such as Perceptron (which will be discussed in following sections), finish training after any "good-enough" hyperplane to divide the data space is found. This approach might result in a boundary with low generalization capabilities, as it might be positioned close to data points of one of the classes while being distant from the others [18]. This is illustrated in figure 2.5.

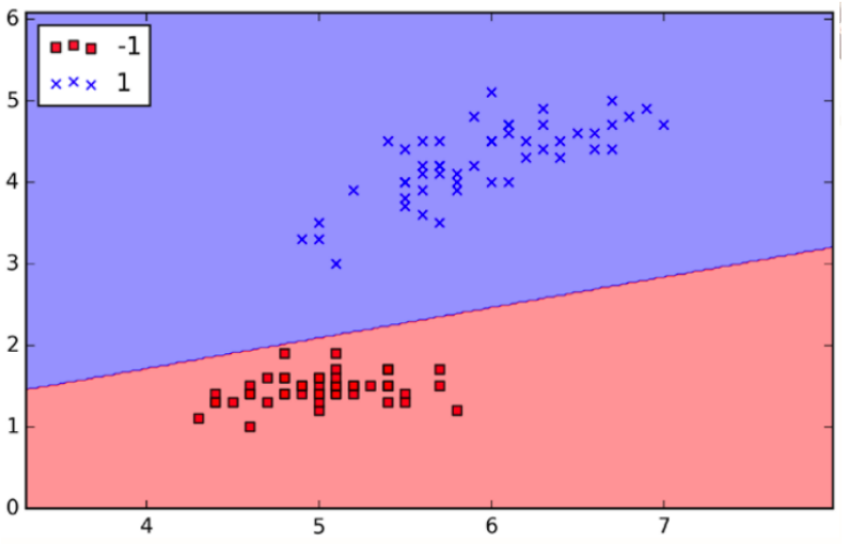

Figure 2.5: Example of a linear class boundary positioned too close to one of the classes

Support Vector Machines (SVM) address this issue by attempting to find the best boundary separating the classes. Such boundary should maximize the margin, i.e. the distance to the closest point on either side. This usually leads to finding a hyperplane exactly "in the middle" between the borders of each of the classes. To find the margin maximizing boundary, the following optimization problem is solved [2]

$$\begin{aligned} \underset{\vec{w}}{\text{minimize}} \quad & \frac{1}{2}||\vec{w}||^2 \\ \text{subject to} \quad & y_i(\vec{w} \cdot \vec{x}) + b - 1 \geq 0, \; i = 1, \ldots, m. \end{aligned} \tag{2.8}$$

As mentioned in the beginning of this subsection, SVMs are capable of solving only linear classification problems. When dealing with multi-class tasks, one can overcome this limitation by training multiple SVM classifiers. One of two following strategies is usually

used:

- *One-vs-One*: Applying this method means training a classifier for each possible pair of classes. During prediction each of these classifiers returns an answer and then a voting mechanism (such as selecting most common class label) is applied to get the final result.
- *One-vs-Rest*: Also known as One-vs-All, this method is realized by training a separate classifier for each class, while objects of all other classes are grouped together under one class label. A class for which the classifier reports the highest confidence score (can be simply the value of $\vec{w} \cdot \vec{x}$) is then chosen as the prediction result.

It is important to note that these strategies are not SVM-specific and can be used with other binary classifiers as well.

In many cases, the classes are not separable by a hyperplane - the data points are "mixed". This problem can be solved by adding a regularization term to the objective function to "punish" missclassification. L2 regularization is used by default in the Scikit-learn SVM implementation:

$$L2 = C \sum_{i=1}^{l} max(0, 1 - y_i w^T x_i)^2 \tag{2.9}$$

In equation, 2.9 C is a free parameter used to express the strength of regularization - higher value of C means higher punishment for missclassification.

The Support Vector Machine algorithm can be extended to solve non-linear problems using so called *kernel trick*. Solving the optimization problem stated in equation 2.8 ultimately leads to a Wolfe dual problem:

$$\begin{aligned} \underset{\alpha}{\text{maximize}} \quad & \sum_{i=1}^{m} -\frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \vec{x_i} \vec{x_j} \\ \text{subject to} \quad & \alpha_i \geq 0,\ i = 1, \ldots, m. \\ & \sum_{i=1}^{m} \alpha_i y_i = 0 \end{aligned} \tag{2.10}$$

where m is the number of samples in the training set, y_i is a class label for sample i and $\vec{x_i}$ is a corresponing feature vector (data sample). As can be derived from the equation, real values of features are actually not needed if the dot product $\vec{x_i} \cdot \vec{x_j}$ is known. We can replace the dot

product with a *kernel* function:

$$K(\vec{x_i}, \vec{x_j}) = \vec{x_i} \cdot \vec{x_j} \tag{2.11}$$

By replacing the dot product in objective function as defined in equation 2.10 with the kernel from equation 2.11 we get:

$$\sum_{i=1}^{m} -\frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j K(\vec{x_i}, \vec{x_j}) \tag{2.12}$$

.

The kernel trick itself is based on the idea that a more complex kernel function using non-linear transformations can be used instead of a dot product (which is often called linear kernel). Kernel functions implemented in the Scikit-learn[2] library for Python include:

- Gaussian radial basis function [21]:

$$K(\vec{x_i}, \vec{x_j}) = exp(-\frac{||\vec{x_i} - \vec{x_j}||^2}{2\sigma^2}) \tag{2.13}$$

  σ being a free parameter in equation 2.13,

- polynomial kernel of degree d [7]:

$$K(\vec{x_i}, \vec{x_j}) = (\vec{x_i}^T \vec{x_j} + c)^d \tag{2.14}$$

  c being a free parameter in equation 2.14,

- sigmoid kernel [14]

$$K(\vec{x_i}, \vec{x_j}) = tanh(\alpha \vec{x_i}^T \vec{x_j} + r) \tag{2.15}$$

  α and r being a free parameter in equation 2.15.

As further explained in section 2.2.2 text classification problems usually include high-dimensional feature vectors. Kernel trick essentially creates a new feature by applying non-linear transformations on existing ones. Already having a dataset with even tens or hundreds of thousands of dimensions, adding one more feature does not significantly increase the amount of information useful for classification. Hence, using linear kernels is often sufficient when working with text data.

[2]https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

2.1.3.4 Probabilistic and Bayesian Classifiers.

Han, Kamber and Pei [9] defines Bayesian (statistical) classifiers as classifiers working with class membership probabilities. Aggarwal and Zhai [1] describe probabilistic and Bayesian classifiers as a mixture of generative models, each modeling data sample probability for a particular class.

This group of classifiers uses Bayes' theorem as a theoretical framework:

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)} \tag{2.16}$$

Equation 2.16 can be used to calculate the posterior probability $P(H|X)$ answering the question: *How probable is the validity of hypothesis H given the evidence X?* In context of classification, the hypothesis under examination is the class membership. The evidence X is then usually a single feature vector describing certain entity or a set of measurements. The question above can be rephrased to *What is the probability of the feature vector X belonging to a certain class C_i?*

According to Han, Kamber and Pei [9], one of the simplest probabilistic methods is the *naïve Bayesian classifier*. During prediction, this classifier simply calculates the posterior probability using Bayes' theorem, as defined in equation 2.16, for all classes in the dataset and selects the class having the highest probability. The denominator $P(X)$ can be excluded from the calculation as it is constant for all classes. $P(H)$, or more descriptively $P(C_i)$, is simply the share of samples belonging to the given class contained in the training set (or can be known a priori).

To simplify the calculation of $P(X|C_i)$, Naïve Bayes assumes feature independence, hence the name *naïve*. Given that this assumption is true, the computation can be reduced to a simple product of probabilities:

$$P(X|C_i) = \prod_{k=1}^{n} P(x_k|C_i) \tag{2.17}$$

In equation refeq:prod, x_k represents the value of feature on k-th index from a n-dimensional feature vector. Calculation of the conditional probability $P(x_k|C_i)$ depends on the feature type. Simply counting the number of value occurrences is sufficient categorical features. When dealing with continuous variables, one must usually model the Gaussian (or other) distribution describing the process of values' generation.

Although Naïve Bayes algorithm is simple in principle and limited by the feature independence assumption, in certain applications it performs comparatively good to more complex

decision tress or neural networks [9].

2.1.3.5 Perceptron and Neural Networks. In section 2.1.3.3, we briefly introduced the idea behind linear classifiers based on finding a hyperplane separating the classes, usually represented by a vector of weights $\vec{w}$. Equation 2.7 shows that if such a hyperplane is found, a binary classification problem can be solved by calculating a dot product between the vector of weights an a data sample and applying a hypothesis function.

In 1943, McCulloch and Pitts [15] proposed similar approach as a simplified model of a neuron cell. Authors described a neuron as a logical gate providing binary output (the cell either fires an impulse on the axon or not). This output depends on a combination of the input signals received through dendrites exceeding a certain threshold. Graphical representation of the McCulloch-Pitts neuron model is shown in figure 2.6

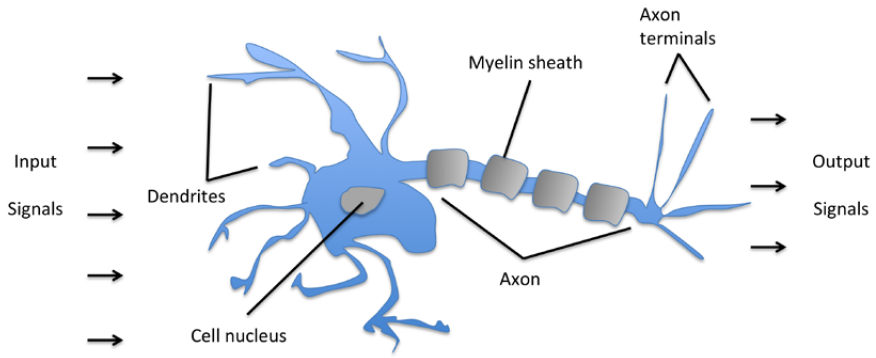

Figure 2.6: Model of a neuron cell as proposed by McCulloch and Pitts [15].
Source: Raschka [18]

Based on this idea of neuron representation, Rosenblatt [19] proposed a model for learning optimal input weights capable of training binary linear classifiers. Equation 2.7 is used for prediction although being interpreted in slightly different terms: the product of weights and a vector of features is considered a weighted sum of neuron inputs. The $signum()$ function is used as an *activation function* representing the threshold for neuron activation; if the neuron is activated, positive class label is assigned to the data sample and the negative label is used if the neuron is not activated.

Raschka [18] summarizes the Perceptron algorithm in modern terms by following steps:

1. Initialize the weights (set to 0 or a small random number for example).
2. For each object (feature vector) $\vec{x}$ in the training set:
   (a) Compute the output y_{real} using equation 2.7.

(b) Update the weights.

where the update process is defined by equation 2.18:

$$\vec{w}_{new} = \vec{w}_{old} + \eta(y_{expected} - y_{real})\vec{x} \tag{2.18}$$

where the expected class label $y_{expected}$ should be known beforehand because classification is a supervised learning problem and the learning rate η is a free parameter. Given that class labels -1 and 1 are used, the difference between the expected and the real output can be -2, 2 or 0.

Agrawal and Srikant [2] explain the Perceptron learning method using geometry and the fact that there is a relationship between a dot product of two vectors and the cosine of the angle between them. They divide the equation into tree rules:

- If the predicted label is 1 and the expected label is -1 the angle between the feature vector and the vector of weights is less than 90°. Feature vector has to be subtracted to increase the angle.
- If the predicted label is -1 and the expected label is 1 the angle between the feature vector and vector of weights is greater than 90°. Feature vector has to be added to decrease the angle.
- Weights do not change if the sample is classified correctly.

A standalone unit trained by the Perceptron algorithm is very limited in terms of classification performance and application. Its importance lies in the fact that multiple perceptrons can be combined into so called *multi-layer perceptron* - a type of a *feed-forward neural network*. Han, Kamber and Pei [9] define the neural network as a set of interconnected input/output units. Each connection in this set has a weight that should be adjusted during the training process, so the network as a whole is capable of predicting a class label (or a continuous value).

Han, Kamber and Pei [9] talk about following advantages of using neural networks:

- tolerance to noisy data,
- ability to classify patterns that the network has not seen during training,
- low domain knowledge requirements,
- given enough units, they can approximate any function with high level of accuracy,

- training process can be easily parallelized.

On the other hand the authors also mention significant disadvantages:

- long training times,
- high number of configuration parameters defining the structure of the network,
- low level of interpretability (as opposed to decision trees for example).

In a feed forward neural network, the input/output units are organized into multiple layers in such a way that each output from one layer is connected to each input of the following one. The first layer in this cascade is called the *input layer*. The number of units or neurons in the layer corresponds to the number of features. The last layer is the *output layer*. In classification each output usually represents one class label. The layers inbetween the input and output layers are called *hidden* layers. For many classification problems using just one hidden layer is sufficient [9].

Each unit in the neural network is a Perceptron-like neuron, with one significant difference being the activation function. In the introduction to the Perceptron algorithm the signum function was used to calculate the final output. For neural networks more complex non-linear functions are chosen. Logistic function already defined in equation 2.2 and rectifier (defined as $f(x) = max(0, x)$) are common options [9, 18].

During prediction using a trained neural network an output value is calculated for each neuron in the first layer. This output is then passed to each neuron in the second layer and the process repeats until the output layer is reached. Finally, the class represented by the highest value output is chosen [18].

The *backpropagation* algorithm is used to train a feed-forward neural network. Similarly to a single perceptron unit, first the output is computed using current weights. Then the error is calculated using following equation:

$$err_j = y_{real}(1 - y_{real})(y_{expected} - y_{real}) \tag{2.19}$$

In equation 2.19, y_{real} is the output computed by the neural network and $y_{expected}$ is the expected value (for example 1 if the neuron was expected to "fire") [9]. This error is then propagated backwards and can be calculated for each neuron as:

$$err_j = y_j(1 - y_j) \sum_k err_k w_{jk}, \tag{2.20}$$

where y_j is the real output of given neuron j, err_k is the error of neuron k belonging to the following layer, and w_{jk} is the weight of the j-k neuron connection [9].

Finally new weight values are calculated:

$$w_{ij,new} = w_{ij,old} + \eta err_j y_j \tag{2.21}$$

As in case of a single Perceptron neuron, η is a free learning rate parameter usually set to a value between 0.0 and 1.0 [9].

2.2 Natural Language Processing

Bird, Klein and Loper [4] define *Natural language processing* (NLP) very broadly as any kind of natural language manipulation utilizing computers. According to the authors, NLP covers anything from simple use cases, such as counting word frequencies, to complex tasks, such as understanding the meaning of a written text. Sarkar [20] talks about NLP as a field of computer science, engineering and artificial intelligence focused on creating systems of interaction between machines and natural languages, and being closely connected to fields like computational linguistics and human-computer interaction. The author also explains the term *natural language* as a language developed through natural use and communication as opposed to artificially created languages, such as mathematical notations or programming languages. Aggarwal and Zhai [1] use the term *text data mining*. Concept of *data mining* was already defined in section 2.1, with key difference being the focus on text data. For the purpose of this thesis, text data mining is understood as the field covering more complex NLP tasks.

In the rest of this section, we will focus mostly on these non-trivial tasks while skipping some common text manipulation methods, such as regular expressions. The section is structured similarly to the previous one: we provide a survey of tasks solved by NLP methods and then focus on text classification specifics.

2.2.1 Survey of Problems Solved by Natural Language Processing methods

This section covers some of the more complex task solved by NLP methods. Some of them were already covered in section 2.1.2 as they represent problems that are solved for other types of data as well.

2.2.1.1 Named Entity Recognition.

Named Entity Recognition (NER) is a part of larger group of problems often called *Information Extraction*. Aggarwal and Zhai [1] say that the goal of the information extraction process is to find structured information in unstructured or semi-structured text. This structured information can be either presented to a user or processed by other software systems. Bird, Klein and Loper [4] also agrees with the definiton noting that although we are still a long way from general-purpose algorithms capable of extracting meaning from text, being able to answer some very specific questions can be still very useful.

Named Entity Recognition itself can be described as the process of finding and categorizing named entities in a text document [1, 20]. Aggarwal and Zhai [1] defines *named entities* as a sequence of words representing a specific real-world entity. Although entity categories are very application and domain dependend, Sarkar [20] mentions some of the universal entity types useful in most contexts: *person*, *organization*, *location*, *date*, *time* or *amount of money*. One of the uses of NER is that it serves as a basis for *relationship extraction* - another important task belonging to the information extraction family of problems.

One of the simplest approaches to named entity recognition is using a set of simple *if-then* rules usually constructed by domain experts. These rules are based on certain properties of examined words. One can for example have a rule stating that if a token comprises only of upper-case letters it is potentially an acronym. Or that a word starting with an upper-case letter followed by only lower-case letters is a name of a person, organization or place.

Modern NER uses machine learning methods such as classification to discover named entities. One of key differences between common classification methods as described in section 2.1.3 is that in case of NER usually a sequence of objects is classified instead of one standalone entit[1]. Advanced techniques such as *recurrent neural networks* are used to solve this type of classification problems [8].

2.2.1.2 Topic Modeling.

Topic modeling is the process of discovering key topics described in a text document. It is closely related to the problem of dimension reduction introduced in section 2.1.2.6.

One of the biggest challenges encountered when applying machine learning methods on text is the fact that many of these techniques are usable only for numeric data. Therefore, there is a need for algorithms transforming text into such numeric representation.

One of the most common approaches for solving this problem is *Bag-of-Words*. It is

based on creating vectors of word frequencies (alternatively more complex metrics can be used). Intuitively, one could utilize such vectors to find most frequent words or groups of words to analyze document topics. This approach has a number of advantages including easy similarity measurement and, as explained in following sections, these Bag-of-Words vectors can be used as features for classification and clustering.

On the other hand, this method has some significant issued that need to be addressed. In general, resulting word frequency vectors are high-dimensional, even relatively small datasets can contain tens of thousands of different words. Directly related to the topic modeling is the issue of polysemy (same word having different meanings depending on context) and synonymy (different word having the same or very similar meaning). Using Bag-of-Words, a computer system might consider documents about a computer mouse and a mouse "the animal" as closely related and on the other hand consider documents about cars and automobiles as completely different.

Latent Semantic Indexing is one of the techniques addressing the problems of polysemy and synonymy. It uses Singular Value Decomposition (SVD) to factorize matrices into a product of three matrices. This technique is also used for dimension reduction and feature extraction [1, 20].

In Latent Semantic Indexing, the text dataset (also called *corpus*) is represented by a matrix D of d columns, each representing one document, and t rows, each representing a unique word or other token. Each value in this matrix can be then determined by the number of occurrences of a given word in a given document. This matrix is then factorized according to equation 2.22:

$$D = U\Sigma V^T \tag{2.22}$$

In equation 2.22, U a V are both orthonormal matrices and Σ is a diagonal matrix containing positive real values [11].

Results of SVD are then used to create a matrix of rank k to approximate the original matrix using fewer dimensions. In LSI, this new matrix purposefully represents different topics mentioned in the document. Effectively, after LSI application, each document is represented as a combination of topics (each topic having certain share or weight) instead of word frequencies [1].

From a certain point of view, topic modeling can also be understood as application of soft clustering methods. Clusters represent different topics and documents can belong to multiple

clusters with certain degree of membership.

2.2.1.3 Sentiment Analysis. Sarkar [20] distinguishes between two types of textual information: factual (objective) and opinion based (subjective). Documents such as social media posts or product reviews contain large amounts of subjective data. They express judgment, beliefs, emotions and feelings of the author. During *sentiment analysis* (also called *opinion mining*), as Sarkar [20] defines it, we extract this information from a text document using methods of machine learning, linguistics and NLP. Data obtained during the process can be used to calculate polarity - expression of a positive, negative or neutral sentiment.

Results produced by sentiment analysis are useful for businesses, governments or political figures as they can be used to effectively analyze how general public perceives products, services, events, government policies or even political ideologies.

Sentiment analysis can be applied on different levels. Polarity can be computed for a single sentence, paragraph or the document as a whole. According to Sarkar [20] two main approaches are used in sentiment analysis:

- supervized ML methods (classification or regression)
- unsupervized lexicon-based methods

If a set of training documents with known sentiment labels or numeric ratings is available, one can train a classification or regression model capable of predicting polarity of a previously unseen document. It is assumed that the document expresses opinion on a single entity provided by single person [1].

Key challenge of any text classification problem is choosing how the document is transformed into a numeric representation. This problem is described in detail in section 2.2.2 and further studied during the experimental phase of the thesis.

Unsupervised methods use pre-built lists of words usually expressing a subjective opinion. For example, words such as *excellent* or *great* are associated with a positive sentiment while *poor* or *underwhelming* are connected with a negative judgment. Taking negatives into account is also important: *great* expresses positive sentiment while *not great* or *isn't great* signals a negative opinion [1]. The SentiWordNet[3] database is often used for unsupervised sentiment analysis.

[3]https://github.com/aesuli/sentiwordnet

2.2.1.4 Clustering. Clustering, i.e. finding groups of similar entities in a data set, has already been discussed in section 2.1.2.4. Most authors agree that same methods as in case of other types of data can be used to cluster documents [1, 20, 4]. As in case of supervised sentiment analysis, main challenge of text clustering is representing the data in a numerical form. One can use techniques same or similar to those used during classification to solve this problem.

Aggarwal and Zhai [1] say that text clustering is a dual problem. Using the matrix representation of a text dataset introduced in section 2.2.1.3, clustering can be performed either on rows (words) or columns (documents). Words occurring in the same document can be put into one group and analogously documents containing the same word can also belong to the same cluster. Moreover, results of one clustering problem can be useful when solving the other.

There are clustering methods build on this close relation. One of them is *Frequent Word Patterns* mining. Similar to pattern mining described in section 2.1.2.3, this process detects frequent word patterns in the dataset and uses them to define different clusters. Then individual documents are examined and assigned to one or more groups.

Some of text clustering application are also described in [1]:

- organization and browsing: hierarchical clustering methods can be utilize to discover a system of document organization,
- corpus summarization: by creating clusters one can detect and describe main groups of documents contained in a data set,
- classification: information about cluster membership can be used to classify documents.

2.2.1.5 Classification. Problem of classification is discussed in detail in sections 2.1.2.1 and 2.1.3. As already mentioned in sentiment analysis and text clustering review, same methods can be utilized to classify text as any other type of data. We mentioned that the key problem of text classification (and also of text regression or text clustering) is transforming text into a numeric representation. In this thesis, we study text representation in depth, experimentally comparing multiple approaches.

2.2.2 Text Vectorization Methods

In section 2.2.1, we mentioned multiple times one of the most important problems encountered when applying machine learning methods on text data. Techniques used for classification, regression or clustering expect numeric input. Therefore, text documents have to be transformed into vectors of numeric features. This process is often referred to as *vectorization*. Referring back to the knowledge discovery process introducted in the beginning of the chapter, vectorization can be described as a specific implementation of the data transformation process.

This part of thesis examines multiple approaches to vectorization. We start with traditional Bag-of-Words family of algorithms and then study more novel methods: embeddings and graph representations. In the experimental part, Bag-of-Words and graph-based methods combined with different classification methods are compared in terms of classification accuracy.

2.2.2.1 Bag of Words Methods. The history of the term *Bag of Words* goes back at least to year 1954, when Zellig Harris used the phrase in his article titled *Distributional structure* [10]. In context of ML and NLP, Bag of Words represents a family of text representation methods transforming the text into a high dimensional vectors, with each dimension representing one word in the whole text data set. Values in these vectors are then based on analyzing word occurrences in a given document.

The simplest technique in this group mentioned by Aggarwal and Zhai [1] is creating a vector of binary values: 1 if a word is present in the document and 0 otherwise. More informative approach is counting word occurrences in the vector. Raschka [18] describes it as a two step process:

1. Construction of a vocabulary of unique words from the corpus.
2. Transformation of documents into feature vectors containing numbers of occurrences of the word in a given document.

Although this method preserves more information compared to the binary representation, it still comes with a significant issue: when it comes to classification there are many words whose occurrence frequency is similar for all classes. Therefore, these words are not useful in the process of distinguishing between the classes.

Stop words are one such group. They can be defined as word used frequently in a certain language by every person. This usually includes articles (*the, a, an*), conjunctions (*and, or*) or prepositions (*to*). Lists of stop words for many languages are available in NLP libraries such as NLTK or can be easily found on the web.

One possibility how to process stop words is to remove them from the document. Another option is replacing word frequency in the Bag of Words vectors with more complex metric taking word relevancy into account. Given that this metric does not rely on any prior knowledge about common words used in a language, the second method can be considered more universal, but also more adaptable to a specific data set.

Common word frequency replacement is the *term frequency - inverse document frequency* (*Tf-Idf*). As Ramos [17] describes it, for a given word w and document d it can be calculated as:

$$w_d = f_{w,d} \cdot log \frac{n_d}{1 + f_{D,w}} \tag{2.23}$$

where:

- $f_{w,d}$ is the frequency (number of occurrences) of the word w in document d,
- n_d is the total number of documents in the dataset,
- and $f_{D,w}$ is the inverse document frequency - number of documents containing word w.

As already mentioned, one of the biggest disadvantages of the bag-of-words methods is that the resulting feature vectors are high-dimensional. Every word in the dataset is represented by one dimension even if it isn't present in the document. Various dimension reduction methods, that we already discussed in sections 2.1.2.6 and 2.2.1.2, can be used to compress the dataset.

Important extension implemented in the Scikit-learn Python library is the n-gram vectorization. Instead of using a single word as a bag-of-words vector dimension, one can also choose to use groups of n following words called n-grams. This approach allows the user to analyze phrase at the cost of even higher dimensionality.

2.2.2.2 Word and Document Embeddings. On a word level, Bag-of-words methods represent each word as a sparse, high-dimensional vector in which only one index has a non-zero value. One of the disadvantages of this approach is that is hard to impossible to compare the meaning of different words, i.e finding related terms or opposites.

In their survey paper, Zhang et al. [24] describe an alternative method called *word embeddings* (also known as *distributed term representations*). Word embeddings encode each word as a dense, low-dimensional vector of continuous values. In practice usually hundreds of dimensions are used. This method is able to capture semantic and syntactic similarities of words and also apply basic algebraic operations to search for a word with given meaning. Authors show this on an example: $wv(\text{"}Madrid\text{"}) - wv(\text{"}Spain\text{"}) + wv(\text{"}France\text{"})$ should result in a vector close to $wv(\text{"}Paris\text{"})$. To achieve this, word embeddings use the context in which the words occur to construct the vector representation, the assumption being that a word can be identified by it's surroundings.

Inducing word embeddings is an unsupervised learning task that requires large text datasets to provide quality results. Zhang et al. [24] talks about $word2vec$ as the most popular method. This model uses a 3-layer neural network to learn word representations. This network is trained to solve one of two tasks:

- predicting surrounding context given a current word (*n-skip-gram*);
- predicting the current word given the context - preceding/following words (*continuous bag-of-words*).

The word embedding itself is then a byproduct created by the hidden layer when solving the classification task.

Until now only word embeddings were discussed. As the name suggest, they represent a single word. Le and Mikolov [12] propose a method how the *word2vec* method can be generalized to also create sentence or document embeddings. If we took the continuous bag of words approach for example, and added a new input representing the document or paragraph (can be a bag-of-words representation for example), the document vector is calculated as another byproduct. This principle is illustrated in figure 2.7.

We do not further examine this method in this thesis, mostly due to time limitations.

2.2.2.3 Word-Graph. In their papers, Violos, Tserpes, Psomakelis, Psychas and Varvarigou [22] and Castillo, Cervantes, Vilariño, Báez and Sánchez [5] examine another approach to text vectorization. They use directed graphs to represent the text. Each node in this graph structure corresponds to one word in the document. Connections between words are vicinity based - an edge connects the words if they can be found close to each other in the text. Words are considered close if they can be captured in a frame of n following words.

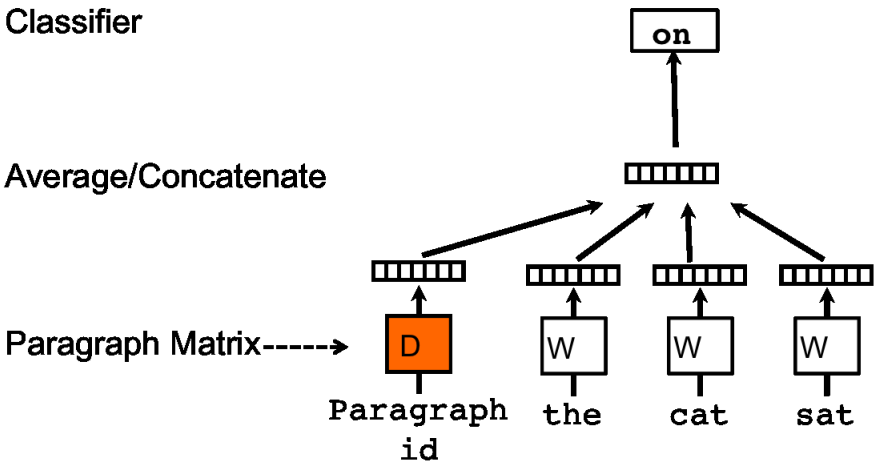

Figure 2.7: Model of a continuous bag of words classifier creating a paragraph embedding. Source: Le and Mikolov [12]

Edge orientation depends on the order of the words in the document. An example of such graph is depicted in figure 2.8.

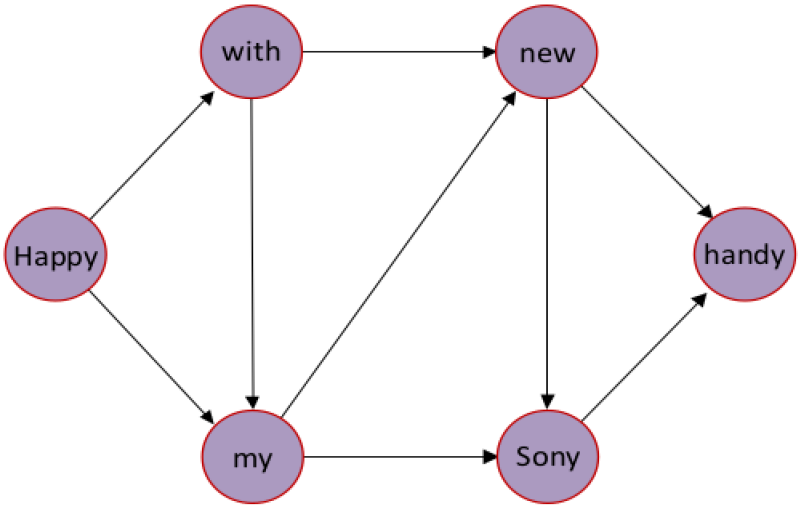

Figure 2.8: Construction of a word-graph for document: *Happy with my new Sony handy*, using frame size of 3. Source: Violos, Tserpes, Psomakelis, Psychas and Varvarigou [22]

Aisopos, Tzannetos, Violos and Varvarigou [3] also propose an alternative approach, using character n-grams (sequences of n following characters) as graph nodes. Both options were tested on Twitter sentiment analysis, with the word-graph approach providing better classification accuracy.

Authors utilize these graph structures in multiple ways. Castillo, Cervantes, Vilariño, Báez and Sánchez [5] use centrality measures to detect most important words in the documents. These are then used to construct a word frequency vector similarly to the bag-of-words

approach. From a certain perspective, we can talk about feature selection. On the other hand, Aisopos, Tzannetos, Violos and Varvarigou [3] and Violos, Tserpes, Psomakelis, Psychas and Varvarigou [22] use similarity metrics to compare document graphs and class graphs, created by merging all document graphs belonging to a given class. Single document is compared with all class graphs in the training set and the similarity metric values are then used as input to train a classifier.

In this thesis, we examine the latter approach in depth. We use *containment similarity* metric to compare the graphs:

$$CS(G_D, G_C) = \frac{\sum_{e \in G_D} \mu(e, G_C)}{min(|G_D|, |G_C|)} \tag{2.24}$$

In equation 2.24, the similarity between a document graph G_D and a class graph G_C is calculated as the number of common edges divided by the number of edges of the smaller graph. Here, $\mu e, G_C$ is a membership function returning 1 if the edge e belongs to graph G_C and 0 otherwise.

Violos, Tserpes, Psomakelis, Psychas and Varvarigou [22] test also more complex similarity metrics, all options providing similar results. Containment similarity is therefore chosen for the experiments; albeit simple, it does not significantly degrade the classification performance.

2.2.3 Text Preprocessing

Before text documents can be vectorized and a classifier trained, usually a set of preprocessing actions has to be performed to increase the "tidiness" of the data and transform it into a suitable form.

This section shortly covers preprocessing steps most relevant to this thesis: tokenization, stop-words removal and stemming.

2.2.3.1 Tokenization. All vectorization methods mentioned in the previous section require the document to be split into smaller units, usually words. Tokenization can be defined as the process of extracting smaller units of text called tokens from a text document. A token can be specified on multiple levels: it can be a single character, a word or a sentence. After extraction, tokens are usually stored in a list-like structure provided by chosen implementation programming language [20].

One of the simple approaches to tokenization is creating a token pattern using regular expressions. This pattern can be then matched against a text document. Most modern programming language provide this functionality "out of the box".

2.2.3.2 Stop-Words Removal. The term *stop-words* was introduced in section 2.2.2.1. It describes words that are used almost by every speaker of a given language. They usually do not provide much value when solving classification or clustering problems. Removing stop-words from the list of tokens might be useful in many scenarios both in terms of time and quality of results.

Lists of stop-words can be found on the Internet or even as a part of natural language processing libraries. The process of stop-words removal is usually as simple as examining every word extracted from the document and checking the list of stop-words if it contains the current word.

2.2.3.3 Stemming. Stemming is the process of removing word suffixes (and also prefixes in some cases), keeping only the main morpheme (*stem*) of the word for further manipulation. Figure 2.9 illustrates how one stem provide a basis for multiple words.

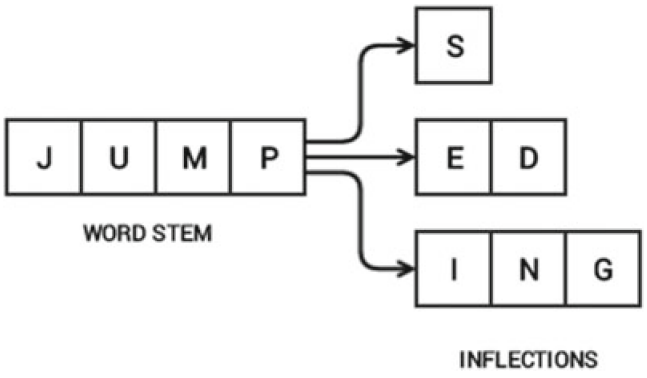

Figure 2.9: Stem "JUMP" can be used to construct multiple words. Source: Sarkar [20]

In morphologically rich languages such as Slovak, the stem carries the meaning of the word. Different suffixes are then used to implement the grammar of the language. Stemming allows us to ignore grammar and focus on the meaning of the words. In many use cases, the grammatical difference between the words *jumps* and *jumped* might be of little to none significance. We might be simply interested in the stem *jump* without distinguishing between the tenses.

For Slovak, a ready-made stemmer based on a set of regular expression rules to identify

and remove the stems can be used. This method is simple, but it might not be able to deal with cases where the stem itself ends with a combination of characters that could be missclassified as a suffix.

3 Methodology of the Automatic Author Identification

In this thesis, we implement a system for automatic author identification, given a finite set of possible authors. More specifically, we are interested in identifying a politician making a speech in Slovak national parliament. As already mentioned in previous sections, this problem ultimately leads to text classification. The process of solving such a task can be described as a pipeline consisting of 5 steps:

1. obtaining training data,
2. filtering,
3. preprocessing,
4. vectorization,
5. classification.

In section 3.2, we describe the format of training data and how it was obtained. This section also covers the reasons why certain data points are removed from the dataset.

In our case, the preprocessing step can be further divided into tokenization and optional stemming. Both of these topics are already covered in detail by previous sections, hence they are not further analyzed.

This section is focused on topics of vectorization and classification. As mentioned, multiple approaches can be used to train classification models. These approaches are not capable of processing raw textual data. Therefore, one of vectorization methods has to be used to transform text documents into a set of numerical features.

3.1 Selection of the Best Approach

In the thesis, we perform multiple experiments to evaluate some of the options for solving a text classification problem and select the best performing one. We examine three text vectorization methods: two variants of bag-of-words (using either word count or term frequency-inverse document frequency metric) and the word-graph method. When it comes to classification itself, we compare 4 models discussed in section 2.1.3: Decision Tree, SVM, Naïve Bayes and Multilayer Perceptron.

Furthermore, multiple hyperparameter combinations are tested. Hyperparameters represent settings that have to be set by the user before the training starts. For example, in case of

SVMs one can set the kernel function and regularization strength. Full list of hyperparameters tuned during the experiments can be found in table 3.1. Two aspects are further considered

Method	**Hyperparameter**	**Explanation**
Tokenizer	preprocessor	function to preprocess the text documents with
Tf-Idf Vectorizer	n-gram range	no. of following words to use as BOW tokens
Word-Graph	window size	max. distance between words considered close
Select K Best	k	no. of best features to select
SVM	kernel C	kernel function regularization strength
Decision Tree	N/A	N/A
Naïve Bayes	N/A	N/A
ML Perceptron	activation hidden layers	neuron activation function structure of hidden layers (neuron counts)

Table 3.1: Hyperparameters tuned for each method used during author identification

when evaluating the method and hyperparameter combinations: classification accuracy (the ratio of correctly classified samples in a test set) and training time.

It is important to note that the training time highly depends on specific implementation and hardware used. Many ML algorithms benefit greatly from being executed on a GPU or other specialized hardware [25]. In our experiments, we use the same hardware and software configuration for each model.

3.2 Training Data

The dataset used for traning the author classifier was obtained from the official website of Slovak National Parliament (NRSR)[4] A Python script[5] was used to traverse the portal and extract the speech transcripts from the HTML code. The speeches with the politician name and time are stored in a file as a JSON list of objects each containing these three attributes. UTF-8 encoding was used to store non-ASCII characters, which are common in Slovak language. Following code illustrates, how a single speech is stored in the file (the transcript is shortened in the example):

```
{
    "speaker": "Ľubica Laššáková",
    "speech": "Ďakujem pekne za slovo. Ja som zaregistrovala štyri otázky, takže
```

[4]http://tv.nrsr.sk/

[5]https://github.com/msvana/nrsr-downloader

ak dovolíte, veľmi rýchlo na ne odpoviem ... " ,
"timestamp": "2018-06-13T18:57:00"
}

In total, the dataset contains 62934 speeches made by 413 unique speakers. The oldest speech was made on May 25 2010 at 10:59 and the most recent one on October 24 2018 at 13:20.

Furthermore, speeches shorter than 250 characters were removed from the dataset under the assumption that these objects could be considered too generic and lower the classification performance.

Because of time and compute power limitations the problem was further reduced to 10 unique politicians. Two variants are further tested:

1. Politicians with similar number of speeches in the dataset; filter is set to select politicians with 300-400 speeches.
2. Politicians with significant differences in number of speeches in the dataset; filter is set to select politicians with 250 or more spechees). After selection, most active politician in the dataset had 1701 speeches.

In both cases, first 10 politicians fulfilling the conditions are selected. The selection is deterministic, no randomization is present.

3.3 Hardware Resources

Paperspace[6] was used as a hardware resources provider for training and testing the classification models. This service provides on-demand virtual machines to execute mostly ML workloads.

The provider allows users to deploy their application using a simple command line tool, significantly improving the workflow. Per-second billing (i.e. paying only for the time the application actually runs) as opposed to a fixed monthly fee is another important advantage.

Various performance options are provided by Paperspace, including virtual machines with a GPU. During experiments we used a C7 option with 12 Intel Xeon E5-2620 CPU cores and 30GB of RAM. This variant provides the best price-to-performance ratio for given purposes. No GPU was used, since tested model implementations can not utilize this type of hardware.

[6]https://www.paperspace.com/

Even though model implementations used during experiments are capable of being executed only on a single CPU core, we are able to fully utilize all provided CPU cores during grid search (see section 3.5). Each CPU core can train one k-fold validation model instance or one variant of the hyperparameter configuration.

3.4 Software for Experiment Implementation

Python was chosen as the main programming language for the experiments. 3 main reasons influenced the choice:

- personal experience with the language;
- large amounts of data analysis, machine learning and text processing libraries; Python is one of the most used languages in the fields of ML and NLP;
- fast prototype development.

When it comes to Python's libraries, most relevant for this thesis are the following:

- *NumPy*[7] allows for fast multidimensional array computations (at least compared to solutions provided natively by Python). The library also serves as a foundation for other data analysis and ML tools.
- *Pandas*[8] is a tool for dataset manipulation. It allows users to import and export data from the Python environment, perform filtering and exploratory data analysis.
- *NLTK (Natural Language Toolkit)*[9] provides tools for solving many NLP tasks including the ones explored in section 2.2.1. Moreover, the library also contains many text corpora one can use to learn and experiment with different NLP algorithms.
- *Scikit-learn*[10] is a library implementing majority of the most popular ML models. All methods examined in this thesis, with the exception of the word-graph, are covered by this library. In addition to these models, Scikit-learn also includes tools for many common preprocessing tasks and other useful utilities simplifying the workflow of developers and analysts.
- *TensorFlow*[11], *Keras*[12], *PyTorch*[13] are libraries focused on solving complex problems

[7] http://www.numpy.org/
[8] https://pandas.pydata.org/
[9] https://www.nltk.org/
[10] https://scikit-learn.org/stable/
[11] https://www.tensorflow.org/
[12] https://keras.io/
[13] https://pytorch.org/

using techniques of deep learning such as advanced neural networks with high number of specialized layers, structured according to various types of architectures. Models of this type require specialized hardware such as GPUs, FPGAs or TPUs to finish training in an acceptable time. These libraries allow the user to utilize these types of hardware and even spread the workload between multiple machines. In this thesis, we do not examine the deep learning approaches in context of solving the author identification problem, and hence do not use any of these libraries.

Paperspace uses Docker images to provide a software environment for applications executed on the platform. To perform our experiments, we use the *paperspace/tensorflow-python* Docker image, which contains all required libraries. The Python script implementing the experiments uses namely Numpy, Pandas and Scikit-learn. The source code of this script is described in detail in section 3.6.

3.5 Grid Search and *k*-fold Cross-Validation

To evaluate multiple combinations of hyperparameter settings, we utilize the *grid search* mechanism provided by the Scikit-learn library. This tool allows the user to train and test a selected machine learning model with different hyperparameter combinations. To use grid search, one must define a grid of parameters - a list of one or more dictionaries, each containing one or more *[parameter]: [list of possible values]* pairs.

When the grid search is executed, it tests all possible combinations of hyperparameter values. During this evaluation, the grid search mechanism is able to utilize multiple cores each training and testing one model instance.

Multiple metrics can be used to evaluate the performance of given hyperparameter combination. In our case, we use classification accuracy - the ratio of correctly classified samples from the testing set. Each combination can be evaluated multiple times. We use 5-fold *cross-validation* in our experiments. When applying this method, the dataset is split into 5 parts, each containing 20% of all available samples. During training, one of these parts is selected for testing, while the rest is used as training data. The process repeats 5 times in total, until each part is used for testing.

K-fold cross-validation allows for more objective evaluation of each hyperparameter combination. According to Raschka [18], cross-validation methods also help to detect over- or underfitting. Overfitting occurs when models are too complex and do not generalize well

to unseen data. Underfitting on the other hand, means that the model is too simple and unable to capture important patterns.

During grid search, detailed data about each examined variant are collected. These metrics include:

- training accuracy of each *k*-fold validation instance;
- mean and standard deviation of the training accuracy;
- testing accuracy of each *k*-fold validation instance;
- mean and standard deviation of the testing accuracy;
- training time of each *k*-fold validation instance (+ mean and standard deviation);
- testing time of each *k*-fold validation instance (+ mean and standard deviation);
- position on a "leaderboard" based on mean testing accuracy;

3.6 Source Code Description

In this section, we describe the source code of the classifier comparison experiment. Figure 3.1 shows the directory structure of the Python project. the */bin* directory contains files which should be used to run the experiments. The application code itself is then located in the */src* folder.

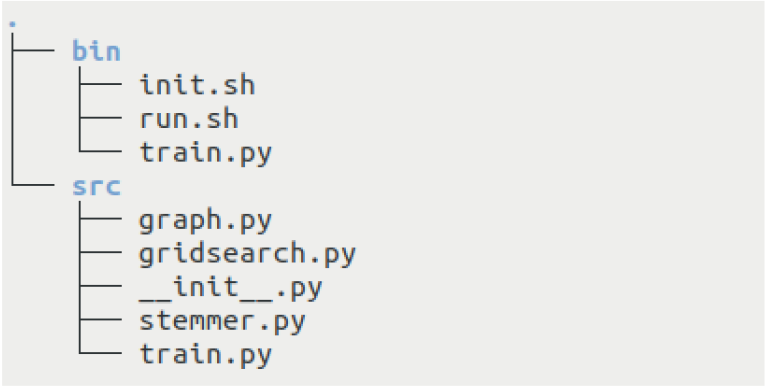

Figure 3.1: Directory structure of the experiment source code

We further focus only on the essential parts of the source code. An interested reader to the publicly available GIT repository [14].

[14]https://github.com/msvana/thesis-author-identification/

3.6.1 Word-Graph Implementation

The *Graph* class defined in the *src/graph.py* file is used to represent the word-graph. Its public interface is designed to serve the purposes of containment similarity metric calculation.

The class code can be seen in figure 4.15. As one can deduct, the word-graph is internally stored as a Python dictionary.

New edges are added to the graph using the `add_edge()` method. A single argument is required - a pair of close words connected by the edge. The direction of the edge is from the first word to the second. In the method's body, we can see that the first word is used as a key for the inner dictionary representation. The value is then a set of all words connected to the "key" word. Using a `set` instead of more common `list` class ensures edge uniqueness. No edge is present twice in the graph.

```
class Graph:

    def __init__(self):
        self._graph = {}

    def get_edge_count(self) -> int:
        total_count = 0
        for node, neighbours in self._graph.items():
            total_count += len(neighbours)
        return total_count

    def add_edge(self, edge: tuple):
        if edge[0] not in self._graph:
            self._graph[edge[0]] = set()
        self._graph[edge[0]].add(edge[1])

    @property
    def structure(self):
        return self._graph

    def get_common_edge_count(self, other: 'Graph'):
        total_count = 0
        for node, neigbours in self._graph.items():
            if node in other.structure:
                intersection = neigbours.intersection(other.structure[node])
                total_count += len(intersection)
        return total_count
```

Figure 3.2: Source code of the Graph class

When calculating the number of edges using the `get_edge_count()` method, we simply iterate trough all graph dictionary items and add the their size to the value of an accumulator

variable (as mentioned, each item in the graph dictionary is a set of words).

We also define a `structure` property that exposes the raw dictionary representation to the outer world. Such property is required to get the number of edges common for two graphs. This functionality is implemented in the `get_common_edge_count()` method. Given some other word-graph, to find the number of common edges we iterate over all key-item pairs of the first graph's dictionary. If the current key is also present in the second graph, we calculate the intersection of corresponding sets and add its size to the value of an accumulator variable.

3.6.2 Word-Graph Similarity Vectorizer

The `WordGraphSimilarity` class implements transformation of text documents into a set of similarity metric values. Figure 3.3 shows the code of public methods. To preserve compatibility with other Scikit-learn components, two methods are provided by the class: `fit()` and `transform()`.

The `fit()` method is in general used to "learn" how to perform a certain transformation. In our case, it constructs the class graphs using close word edges found in all documents belonging to given class. The class graphs are stored in a dictionary using the class labels as keys. During class graph construction we iterate trough the list of documents. If the the class graph for the current document's class is not found in the dictionary, we create a new empty graph (instance of the `Graph` class described above). Then, we simply iterate trough all tokens in the document and add the edges representing word closeness to a corresponding class graph.

After the transformation is "learned", the `transform()` method can be used to actually create a vector of numerical values to represent the document. A word-graph is constructed for each document in the input list. Afterwards, this graph is compared to each class graph created by the `fit()` method by calculating the containment similarity metric as defined in equation 2.24. The vector of all class similarities is then used to represent the document.

Finally, the `__init__()` method allows the user to set the window size - maximal distance between words in the document to be considered as close.

Although not displayed in figure 3.3, an important part of the whole process is tokenization. Inspired by Scikit-learn, we use a simple regular expression tokenizer splitting the document at word breaks (represented by a `\b` token in Python regular expression implementation). Additionally, numbers are ignored under the assumption that they contain mostly

```
class WordGraphSimilarity(base.TransformerMixin):

    def __init__(self, window_size: int):
        self._class_graphs = {}
        self._window_size = window_size

    def fit(self, X, y) -> 'WordGraphSimilarity':
        self._class_graphs = {}

        for i in range(len(X)):
            graph_name = y[i]
            if graph_name not in self._class_graphs:
                self._class_graphs[graph_name] = Graph()
            for edge in self._iter_edges(X[i]):
                self._class_graphs[graph_name].add_edge(edge)
        return self

    def transform(self, X: numpy.ndarray) -> numpy.ndarray:
        X_transformed = []

        for i in range(0, X.shape[0]):
            doc = X[i]
            similarities = []
            document_graph = Graph()

            for edge in self._iter_edges(doc):
                document_graph.add_edge(edge)

            doc_edges = document_graph.get_edge_count()

            for class_graph in self._class_graphs.values():
                class_edges = class_graph.get_edge_count()
                common_edges = document_graph.get_common_edge_count(class_graph)
                similarity = (common_edges / min(doc_edges, class_edges))
                similarities.append(similarity)

            X_transformed.append(similarities)

        X_transformed = numpy.array(X_transformed)
        return X_transformed
```

Figure 3.3: Source code of the Word Similarity Vectorizer

factual data not useful for distinguishing between multiple speakers.

3.6.3 Stemmer

For the purposes of this thesis, a publicly available, MIT licenced Slovak language stemmer[15] created by Marek Šuppa is used. This solution is based on fixed lists of morphological suffixes used for example to express a noun case, a verb tense or to derive a new word. All suffixes defined in these lists are simply removed during the stemming process.

Our observations show that the stemming results are imperfect, but because of time limitations and the fact, that stemming is not the main topic of the thesis, we consider this solution as sufficient.

Use of stemming is further examined during experiments, the sub-goal being answering the question, whether morphology is a significant factor for author identification.

3.6.4 Grid Search Configuration

The *src/grid_search.py* file contains all classifier variants examined using the grid search technique. All configurations are stored in a dictionary having a unique configuration name as a key. The configuration name is a concatenation of the vectorization and classification method names, for example *graph_svc* for word-graph similarity vectorization combined with a SVM classifier.

Each configuration is then defined as a dictionary with two keys. The `steps` key contains a list of steps performed in given order to train and use a classifier. Each step is a pair made up of a step name and an object performing the step.

As shown further, Scikit-learn allows the user to build a pipeline of multiple steps and then use it as a single classifier. The only requirement is that the steps must provide a specific interface, namely the `fit()` and `tranform()` methods such as those implemented by the `WordGraphSimilarity` class.

The second key of a single grid search configuration dictionary is the `params` key. This key contains all hyperparameter values that should be examined by the grid search mechanism. As required by the Scikit-learn `GridSearchCV` class, the hyperparameters are defined as a list of dictionaries, while only a single dictionary is required in this case. The hyperparameter dictionary keys need to follow the [step_name]__[argument] format where the step name depends on the step definitions in described by previous paragraphs and the argument part has to match the corresponding argument passed to the `__init__()` method of the step.

[15] https://github.com/mrshu/stemm-sk

Figure 3.4 shows an example of a single configuration. In this case, we combine the term frequency - inverse document frequency vectorizer with a decision tree classifier. Since the vector created using the tf-idf approach is high-dimensional, we also use the `SelectKBest` feature selector to use only up to 1000 most useful words for classification.

```
'tfidf_tree': {
    'params': [{
        'anova__k': [100, 500, 1000],
        'vectorizer__ngram_range': [(1, 1), (1, 2), (2, 2), (3, 3)],
        'vectorizer__preprocessor': [None, src.stemmer.stem_document]
    }],
    'steps': [
        ('vectorizer', feature_extraction.text.TfidfVectorizer()),
        ('anova', feature_selection.SelectKBest()),
        ('clf', tree.DecisionTreeClassifier())
    ]
},
```

Figure 3.4: Example of a Grid Search configuration

In the parameters part we define that we intend to examine multiple n-gram options (see section 2.2.2, variants with and without stemming and also different number of features selected by `SelectKBest`. In this case, the grid search technique would examine 24 different hyperparameter configurations.

3.6.5 Training Process

The actual training process is implemented in the *src/train.py* file. Before the training starts, the dataset has to be loaded into the memory and preprocessed. This functionality is implemented by the `prepare_data()` function displayed in figure 3.5.

This function loads the dataset from a local JSON file using the Pandas library and separates the data points (speeches) from their labels (speaker names). Following the Scikit-learn notation, the corresponding variables are named `X` and `y`. Afterwards, two filters are applied on the data, the first one being the `filter_short_speeches()` function. As can be seen in the source code in figure 3.6, the function removes speeches that are shorter then a certain fixed value. We use number of characters to define length. This filter is applied under the assumption that short speeches do not contain enough data to reliably identify the author.

The second filter then removes speeches belonging to authors not fulfilling restrictions on number of speeches for a single author. In other words, authors having less or more speeches than defined by given thresholds are removed from the dataset. This filter, implemented by

```
def prepare_data():
    nrsr_df = pandas.read_json(NRSR_DATA_FILE, orient='records')
    X, y = nrsr_df['speech'].values, nrsr_df['speaker'].values
    X, y = filter_short_speeches(X, y, MIN_SPEECH_LENGTH)
    X, y = filter_by_sample_amount(X, y, MIN_SAMPLE_AMOUNT, MAX_SAMPLE_AMOUNT)

    unique_speakers = numpy.unique(y)
    unique_speakers_filtered = unique_speakers[:MAX_SPEAKERS]
    X = X[numpy.isin(y, unique_speakers_filtered)]
    y = y[numpy.isin(y, unique_speakers_filtered)]

    _, count = numpy.unique(y, return_counts=True)
    print('Total speakers:', len(count))
    print('Speakers counts: ', count)

    label_encoder = preprocessing.LabelEncoder()
    y = label_encoder.fit_transform(y)

    X_train, X_test, y_train, y_test = model_selection.train_test_split(
        X, y, stratify=y, test_size=0.1)
    return X_train, X_test, y_train, y_test
```

Figure 3.5: Source code of the prepare_data() function

```
def filter_short_speeches(X: numpy.ndarray, y: numpy.ndarray,
                          min_length: int) -> Tuple[numpy.ndarray, numpy.ndarray]:
    numpy_len = numpy.vectorize(len)
    have_min_len = numpy_len(X) >= min_length
    X = X[have_min_len]
    y = y[have_min_len]
    return X, y
```

Figure 3.6: Source code of the filter_short_speeches() function

the `filter_by_sample_amount()` function, uses advanced indexing options provided by the Numpy library, as can be seen in figure 3.7.

The `numpy.unique()` function returns unique values (class labels in this case) in an array. As a side product, it also provides numbers of occurrences for each unique value and a list of reverse indices. This list contains indices to the list of unique values produced by the function. The value on a given position refers to the unique value present in the original array on the same position. For example, given a list of unique values `[1, 2, 3]` and reverse indices `[0, 0, 2, 1]`, we can reconstruct the original array: `[1, 1, 3, 2]`.

Next, the `numpy.where()` function returns indices at which a certain array fulfills certain conditions. In our case, it returns indices to the list of unique values counts fulfilling the restrictions.

```
def filter_by_sample_amount(
        X: numpy.ndarray, y: numpy.ndarray,
        min_amount: int, max_amount: int) -> Tuple[numpy.ndarray, numpy.ndarray]:
    _, idx, count = numpy.unique(y, return_inverse=True, return_counts=True)
    X = X[numpy.in1d(idx, numpy.where((max_amount >= count) & (count >= min_amount))[0])]
    y = y[numpy.in1d(idx, numpy.where((max_amount >= count) & (count >= min_amount))[0])]
    return X, y
```

Figure 3.7: Source code of the filter_by_sample_amount() function

Finally, the `numpy.in1d()` returns a list of boolean values. The value on a given position is `True` if given item in the first array can also be found in the second array. In the thesis, the function is used to mark positions containing unique values fulfilling the requirements on sample counts. The resulting boolean array can be then used to select values on `True` positions from the list of documents and classes.

Returning to the `prepare_data()` function, after the filters are applied, only a certain number of speakers (10) with their documents is selected for further experimentation. This selection is applied due to limitations of available hardware resources.

Then, the speaker names are replaced by unique numbers using the `LabelEncoder` class from Scikit-learn. Many classifiers can only work with numerical labels. If needed, the label encoder also provides a function for obtaining the original label from its numerical replacement.

The final preprocessing step is splitting the dataset into a training set consisting of 90% of available documents and a testing set made up of the remaining 10% of the samples.

The rest of the the training procedure is straightforward. First, selected grid search configuration is used to create a `Pipeline` object allowing the user and the grid search mechanism to use a sequence of steps as a single classifier. Defined hyperparameter configurations are then tested on the pipeline using the grid search mechanism implemented by the `GridSearchCV` class. The `n_jobs=-1` parameter setting enables parallel training using all available CPU cores. After examination, the best performing hyperparameter configuration is used on the testing set to confirm the results. Finally, full grid search results are stored in a CSV file for later analysis.

3.6.6 Executables

The *bin/* directory contains executable files that should be used to run the exepriment.

Before the training actually starts, the *init.sh* script should be executed in Paperspace

environment to download the Slovak National Parliament dataset and save it on Paperspace's permanent storage mounted on the */storage* directory. A well known *wget* command line utility for GNU/Linux is used for these purposes.

The *train.py* Python script simply executes the `train()` function from *src/train.py*. Separate executable file is created to provide an intuitive way of executing the training process.

Finally, the *run.sh* script allows the user to execute a command inside the Paperspace environment. It serves as a shortcut for the *paperspace-python run* command provided by the *paperspace* package[16] for Python. The user does not have to provide parameters such as machine type (determines available hardware resource) or the path to the project directory that should be uploaded to Paperspace. The command to execute can be defined using the `COMMAND` variable. Two values are used to perform the experiment: first `bash bin/init.sh` to download the dataset (needs to be run only once, even if the experiments are repeated) and then the `python3 bin/train.py` to start training using grid search configuration set by the `GS_CONFIG` constant in *src/train.py*.

[16]https://pypi.org/project/paperspace/

4 Results of Speech Author Identification Experiments

The analysis of experiment results is split into multiple parts. We start with the experiments performed on the dataset containing balanced number of documents for each class. Then, the imbalanced dataset is examined. In both cases we compare best-performing hyperparameter configurations, focusing first on the classifier dimension and then on the vectorizer dimension.

We also analyze the influence of stemming on classification performance. This allows us to see how important morphology is in distinguishing between author.

Next, based on the results, we propose and further analyze a set of modifications to the word-graph vectorization method possibly improving classification accuracy.

All accuracy and training time measurements in this section are mean values obtained using 5-fold cross-validation, i.e. calculated from 5 samples. Full results, including standard deviations, can be found in the annexes.

4.1 Balanced Dataset

As defined in section 3.2, this dataset containes documents from 10 speakers, ranging from 300 to 400 documents per speaker. Exact numbers of documents per class were 316, 347, 393, 348, 348, 377, 332, 310, 302 and 303.

4.1.1 Count Vectorizer Results

Count vectorizer, one of the bag-of-words methods discussed in section 2.2.2.1, represents text documents as vectors of word occurences. As depicted in figure 4.1, best accuracy is achieved in combination with a neural network classifier. The cost of the greater than 90% accuracy is long training time, reaching almost 50 seconds. This is almost 3 times as much as the second slowest model.

The SVM model (with a linear kernel) with accuracy of 83% and training time of 12.5s subjectively provides the best accuracy-time ratio. The decision tree classifier provides good training accuracy (calculated by predicting classes labels of the training set) but very low testing accuracy. This signalizes overfitting - a phenomenon shortly described in section 2.1.2.6

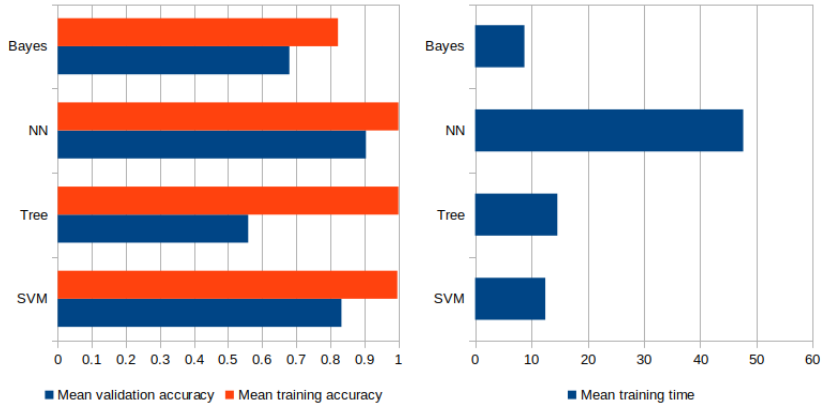

Figure 4.1: Count vectorizer accuracy and training times for different classifiers

4.1.2 Tf-Idf Vectorizer Results

Tf-Idf vectorizer replaces the word count with a more complex term frequency-inverse document frequency metric, taking word relevancy into account (see section 2.2.2.1). As in the case of count vectorizer, figure 4.2 shows, that the best accuracy is provided by a neural network classifier. However, the training time reached almost 9 minutes, which is more than an order of magnitude higher than all other solutions. As before, the SVM provides the best compromise. Its slightly lower accuracy is compensated by significant training time improvement. Characteristics of the decision tree classifier are also similar to those of the count vectorizer solution.

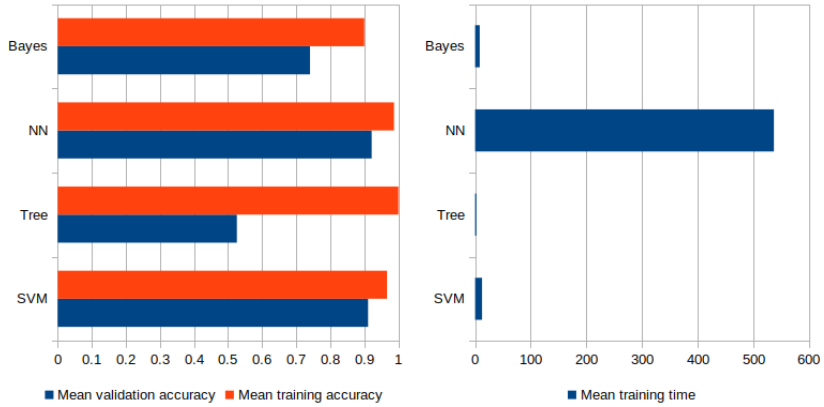

Figure 4.2: Tf-Idf vectorizer accuracy and training times for different classifiers

4.1.3 Word-Graph Vectorizer Results

Figure 4.3 shows the classification results for the new word-graph similarity vectorization method. In this case, the SVM provides the best accuracy, but also takes the longest time to

train. Validation accuracy of the decision tree is poor, it barely reaches 10%. In case of the naïve Bayes classifier, we can also observe grater than 30% difference between the training and validation accuracies, possibly signalizing overfitting. Another notable finding is, that all classifiers provide training accuracy of 1.

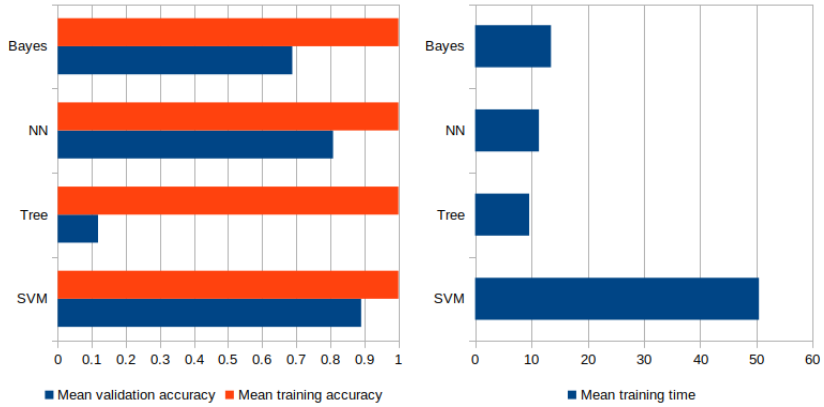

Figure 4.3: Word-graph vectorizer accuracy and training times for different classifiers

For word-graph, one parameter to configure is the window size (max. distance between two words to be considered close). The influence of this factor is analyzed for the best performing SVM classifier. As we can see in figure 4.4, with growing window size the classification performance slightly increases. There is almost linear relationship between the training time and the window size: if the window size grows by 1, the training time increases by approximately 10 seconds.

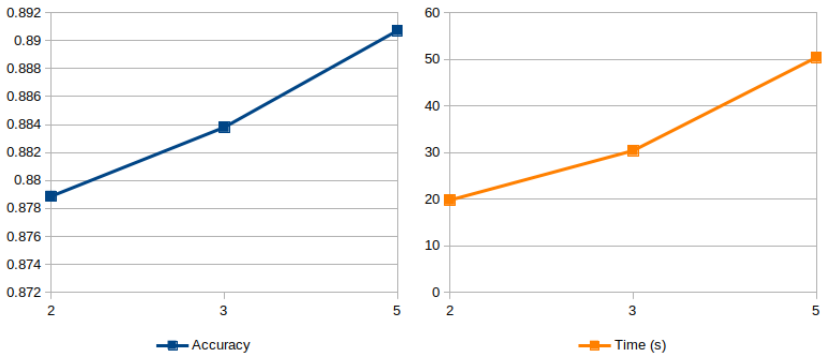

Figure 4.4: Word-Graph vectorizer accuracy and training times depending on window size

4.1.4 Naïve Bayes Results

According to figure 4.5, the naïve Bayes classifier provides best training set accuracy in combination with the word-graph vectorizer. However, it takes the most time to train and the

validation results for unseen data are placed only between the count and tf-idf vectorizers. The latter provides the best validation accuracy and the shortest training time.

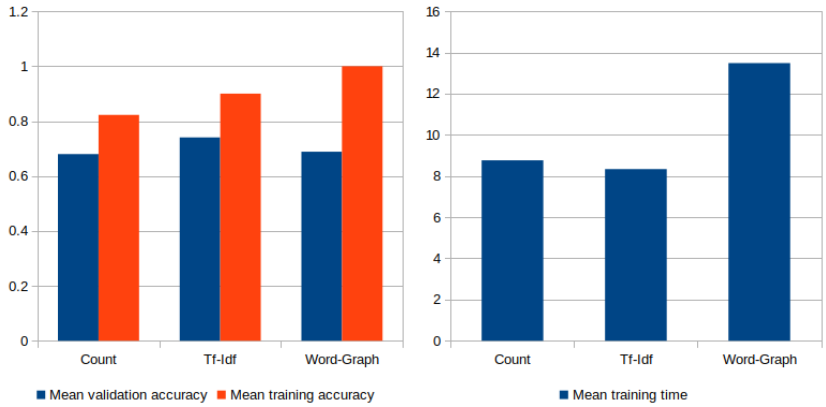

Figure 4.5: Naïve Bayes classifier accuracy and training times for different vectorizers

4.1.5 Neural Network Results

When using a neural network as classifier, the best validation accuracy, approximately 92%, is reached in combination with the tf-idf vectorizer. Figure 4.6 also shows, that the word-graph method provides the lowest performance - validation accuracy just bellow 81%. The training time is highest for the tf-idf vectorizer, reaching almost 9 minutes. Compared to other solutions, this value is exceptionally high. This can be explained by the fact, that for the tf-idf a neural network configuration with 200 hidden layer neurons was chosen as having the best accuracy. Downgrading to 100 neurons, the training time would decrease more than 10 times to 49 seconds, while the accuracy would be only 0.3% lower. Such configuration would still outperform the count and word-graph vectorizers.

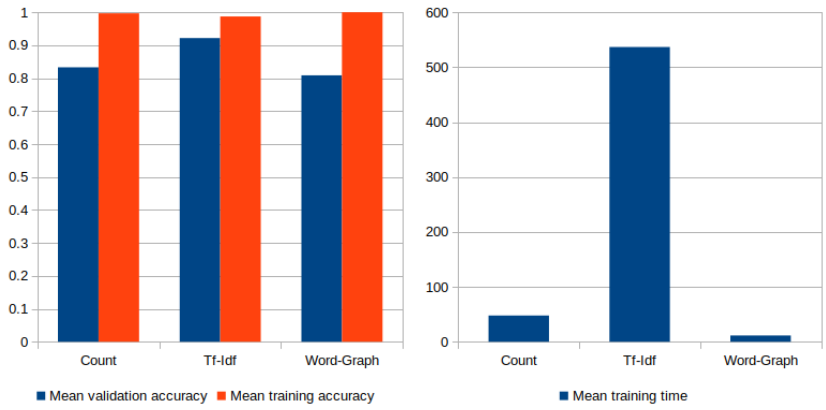

Figure 4.6: Neural network classifier accuracy and training times for different vectorizers

4.1.6 Decision Tree Results

Decision tree classifier provides the lowest accuracy for all vectorization methods. As shown in figure 4.7, it is also the only classifier having lower validation accuracy with the tf-idf vectorizer than with the count vectorizer. Combined with the word-graph approach, the model is practically unusable, providing only 11% validation accuracy. Even though the validation accuracy is very low, the training set measurements show perfect 1.0 score. We can assume the presence of significant overfitting in this model.

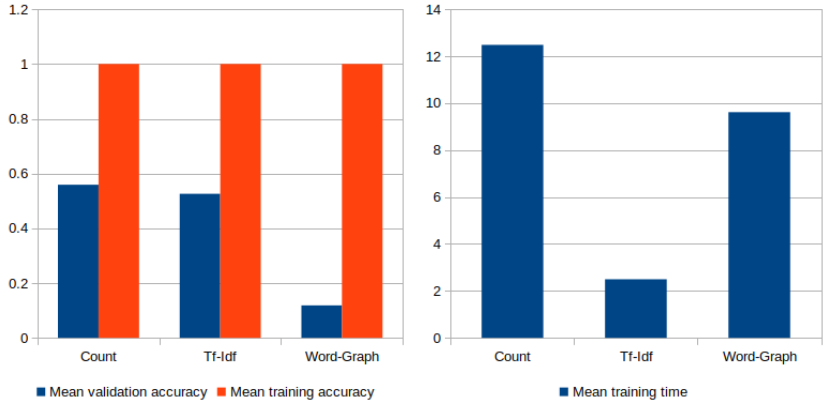

Figure 4.7: Decision tree classifier accuracy and training times for different vectorizers

4.1.7 SVM Results

The SVM classifier provides good accuracy-training time ratio for each vectorization method. Best accuracy is achieved with the tf-idf vectorizer, the difference being just around 1% compared to the count vectorizer on the second place. The word-graph falls behind in both time and accuracy, yet the results achieved are still reasonable. The training time of the word-graph variant could be further improved by lowering the window size at the cost of slightly lower accuracy.

It is important to note, that the word-graph implementation is built "from scratch" in Python as a proof-of-concept. It could be further optimized for example by rewriting parts of the code to a lower-level programming language such as C.

4.2 Imbalanced Dataset

The imbalanced dataset contains speakers with relatively high differences in per class document counts. It is created by selecting first 10 speakers having 250 or more speeches or more.

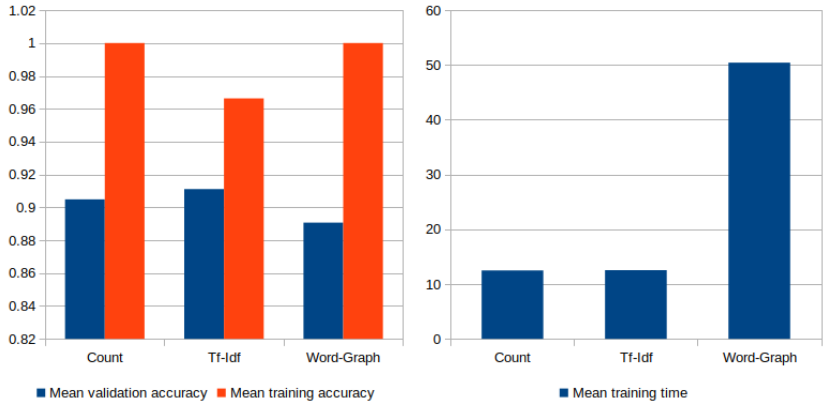

Figure 4.8: Decision tree classifier accuracy and training times for different vectorizers

The resulting set contains classes with 316, 1701, 445, 252, 293, 734, 793, 347, 549 and 253 documents (5430 documents in total, compared to 3376 documents in the balanced dataset).

In this part, we focus on comparing validation accuracies of the balanced and imbalanced datasets to analyze how different methods perform under these worsened conditions. We are not studying the training accuracy, hence do not make any conclusions regarding overfitting.

4.2.1 Count Vectorizer Results

For count vectorizer, both the validation and training times increased for all classifier methods, compared to the balanced dataset. This change is depicted in figure 4.9. Longer training time can be accounted to the higher number of documents in the set. We assume that this factor also improved the accuracy, as the classifiers had more data to train on. Overall, the biggest increase was recorded for the decision tree classifier. The subjectively best accuracy-training time ratio is again provided by the SVM classifier.

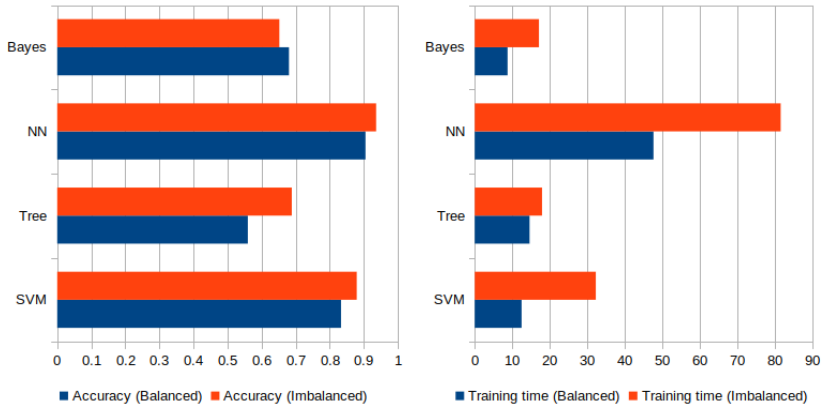

Figure 4.9: Count vectorizer accuracy and training times for different classifiers

4.2.2 Tf-Idf Vectorizer Results

Behaviour of the tf-idf vectorizer is similar to the count vectorizer. According to figure 4.10, there is a significant accuracy improvement for all classifiers compared to the balanced dataset. The neural network classifier again provides the best accuracy at the cost of significantly longer training time. However, the situation is analogous to the balanced set neural network tests. Using less complex structure, the training time is more than 10 times shorter, while only small accuracy decrease is observed (from 94.8% to 94.5%).

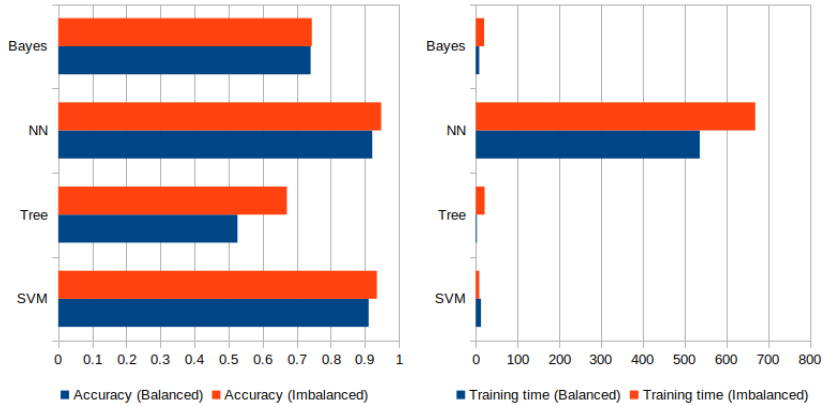

Figure 4.10: Tf-Idf vectorizer accuracy and training times for different classifiers

4.2.3 Word-Graph Vectorizer Results

Compared to count and tf-idf vectorizers, the reaction for the word-graph method to imbalanced numbers of samples per class heads in the opposite direction. The accuracy drops for all classifiers, signalizing the inability of this vectorization approach to cope with such situations. Most notably, according to figure 4.11, the decision tree accuracy drops below 6%, which is much lower than the accuracy provided by a primitive approach of selecting the most frequent class as predicted class label. Moreover, at the same time, the training time is more than 6 times longer.

Only the SVM classifier provides reasonable results even for imbalanced classes.

As can be seen in figure 4.12, the relation between accuracy and window size is also interesting. With growing widow size the accuracy actually decreases. On the other hand, the training time behaves as expected. Similarly to the balanced dataset, the relationship with the window size is almost linear, although the training time is longer in general. This can be explained by higher number of documents in the dataset.

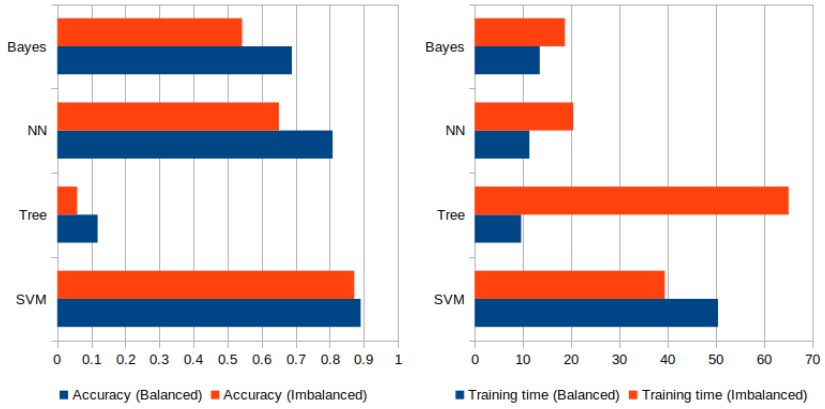

Figure 4.11: Word-Graph vectorizer accuracy and training times for different classifiers

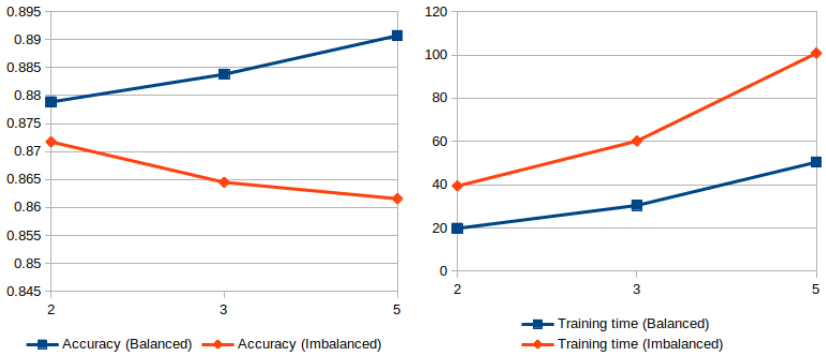

Figure 4.12: Word-Graph vectorizer accuracy and training times depending on window size

4.3 Effects of Stemming

Stemming, i.e. the process of removing morphological suffices from words, was described in detail in section 2.2.3.3. As mentioned, on of the subgoals of this thesis is to examine its effects on the classification process.

To do so, two models are examined in detail: word-graph and tf-idf vectorizers combined with a neural network classifier. The Imbalanced dataset is used to train the models.

Figure 4.13 shows accuracy and training time for both models, comparing the best performing hyperparameter configurations with and without stemming. We can see that for both vectorizers, stemming causes a decrease in accuracy, while significantly increasing the training time (other paramers of the model stay the same).

We can conclude that removing morphological suffices has a small negative effect on classification results. This might be related to the fact that the dataset language, Slovak, is considered a morphologicaly rich language; sentence construction and even meaning of words strongly depends on morphological suffices. Taking longer training times into account, stemming is not recommended for this type of data.

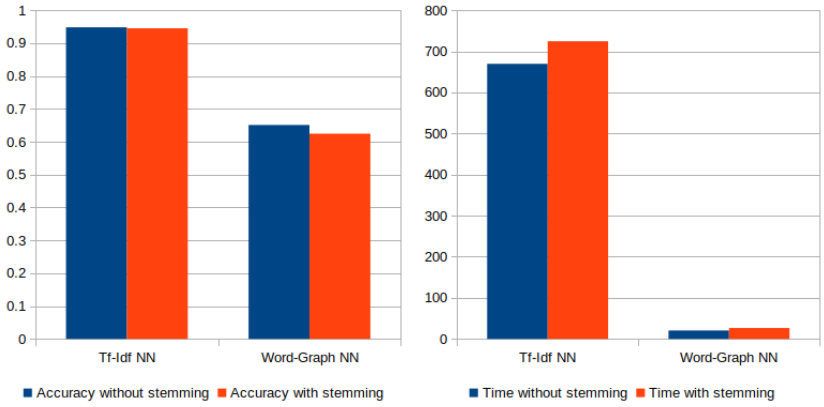

Figure 4.13: Validation accuracy and training times for models with and without stemming

4.4 Word-Graph Modifications

Based on the experiment results, we propose a set of modifications to potentially improve classification accuracy of the word-graph vectorizer. As seen, the performance is low especially for the imbalanced dataset and the decision tree classifier. These modifications are further tested on the same data to evaluate their contribution.

4.4.1 Edge Weights

Violos, Tserpes, Psomakelis, Psychas and Varvarigou [22] propose a set of future directions in word-graph method research. One of these suggestions is adding weights to the graph edges. We propose a simple approach inspired by bag-of-words methods - defining weight of each edge as number of word pair occurrences. This line of thinking leads to a modified similarity metric:

$$CS(G_D, G_C) = \frac{\sum_{e \in G_D} min(w_{e,G_D}, w_{e,G_C})}{min(|G_D|, |G_C|)} \tag{4.1}$$

In equation 4.1, $w_{e,G}$ is the weight of edge (word pair) e inside graph G. Graph size calculation is also changed. Simple edge count is replaced by sum of all weights:

$$|G| = \sum_{e \in G} w_{e,G} \tag{4.2}$$

4.4.2 Feature Scaling

One of possible explanations of poor validation accuracy is the difference between similarity values calculated for training and testing sets. At least one similarity value for a document in a training set is always 1 (for the class graph the document belongs to). This is not the case

for unseen data. We can say, that the training set does not accurately represent the problem. To solve this issue, we propose to scale the similarity values for each document. Two options are further studied:

- interval scaling: similarity values are scaled to spread across the [0; 1] interval;
- normalization: similarity values are scaled to have a mean value of 0.0 and standard deviaton of 1.0

4.4.3 Recurrent Edges

This modification is again inspired by bag-of-words methods. We propose to add recurrent edges to the word-graph, connecting each word with itself. Combined with edge weights, this adds information about word counts to the graph.

4.4.4 Weighted Classes

The containment similarity metric calculated according equation 2.24 can potentially favor classes with higher number of training samples. Our assumption is, that larger amount of samples leads to larger amount of unique edges in the class graph, hence higher chance that a given edge can be found both in the document graph and the class graph. We propose to modify the measurement to also include a "class size" coefficient:

$$CS(G_T, G_C)^* = CS(G_T, G_C)/ln(n_C) \tag{4.3}$$

In equation 4.3, we divide the similarity metric by the natural logarithm of number of documents for given class in the training set. This solution is based on the assumption that word pairs start to repeat for large classes and hence the influence of the class size starts to diminish. Other alternative solutions should be examined in the future.

4.4.5 Modified Graph Implementation

Proposed modifications require significant changes in the source code. Starting with the `Graph` class, the original inner word-graph representation has to be replaced by a new structure. This new dictionary uses strings created by concatenating word pairs as keys and edge weights as values. As shown in figure 4.14, the user can now choose whether the weights should be applied or not. The behaviour of the `add_edge()` method depends on this choice.

In both cases, if an edge is not present in the graph, it is added with weight 1. If the edge already exists, its weight is increased, given that the weighted edges are allowed.

```
class Graph:

    def __init__(self, weighted: bool = True):
        self._graph = {}
        self._edge_cnt = None
        self._weighted = weighted

    def get_edge_count(self) -> int:
        if self._edge_cnt is None:
            self._edge_cnt = sum(self._graph.values())
        return self._edge_cnt

    def add_edge(self, edge: tuple):
        edge_name = '%s:::%s' % edge
        if edge_name not in self._graph:
            self._graph[edge_name] = 1
        elif self._weighted:
            self._graph[edge_name] += 1

    @property
    def structure(self) -> dict:
        return self._graph

    def get_common_edge_count(self, other: 'Graph') -> int:
        total_count = 0
        for node, weight in self._graph.items():
            if node in other.structure:
                other_weight = other.structure[node]
                total_count += min(weight, other_weight)
        return total_count
```

Figure 4.14: Source code of the modified Graph class

Edge count calculation is also modified to match this new graph representation. Instead of summing list lengths, it adds up all the edge weights (values of the graph dictionary). Number of edges shared by two graphs is calculated as a sum of minimum weights. For example if an edge has a weight of 8 in one graph and 5 in the second, 5 is added to the total sum.

4.4.6 Modified Word-Graph Similarity Vectorizer Implementation

The `transform()` method of the `WordGraphSimilarity` class implements the similarity scaling and class weight modifications. The user has to provide the scaling function as a

parameter during initialization. In our experiments we use two functions provided by Scikit-learn - `sklearn.preprocessing.scale` and `sklearn.preprocessing.minmax_scale`. Reccurent edges and edge weights also have to be allowed by setting corresponding parameters of the `__init__()` method.

```
def transform(self, X: numpy.ndarray) -> numpy.ndarray:
    X_transformed = []

    for i in range(0, X.shape[0]):
        doc = X[i]
        similarities = []
        document_graph = Graph(weighted=self._weighted)

        for edge in self._iter_edges(doc):
            document_graph.add_edge(edge)

        doc_edges = document_graph.get_edge_count()

        for class_graph in self._class_graphs.values():
            class_edges = class_graph.get_edge_count()
            common_edges = document_graph.get_common_edge_count(class_graph)
            similarity = (common_edges / min(doc_edges, class_edges))
            if self._weighted_classes:
                similarity /= numpy.log(class_edges)
            similarities.append(similarity)

        X_transformed.append(similarities)

    if self._scaler:
        X_transformed = self._scaler(X_transformed, axis=1)
    else:
        X_transformed = numpy.array(X_transformed)
    return X_transformed
```

Figure 4.15: Source code of the modified WordGraphSimilarity.transform method

Reccurent edges are implemented in the `_iter_edges()` method. If allowed, the initial value of offset for calculating close word indices is set to 0 instead of 1.

4.4.7 Modification Results

The word-graph modifications were first tested on the balanced dataset. Figure 4.16 shows classification accuracy of standalone modifications and their best combination.

In all cases the modifications helped to improve the model quality. Certain modifications actually caused a decrease in accuracy when used separately, for example scaling with

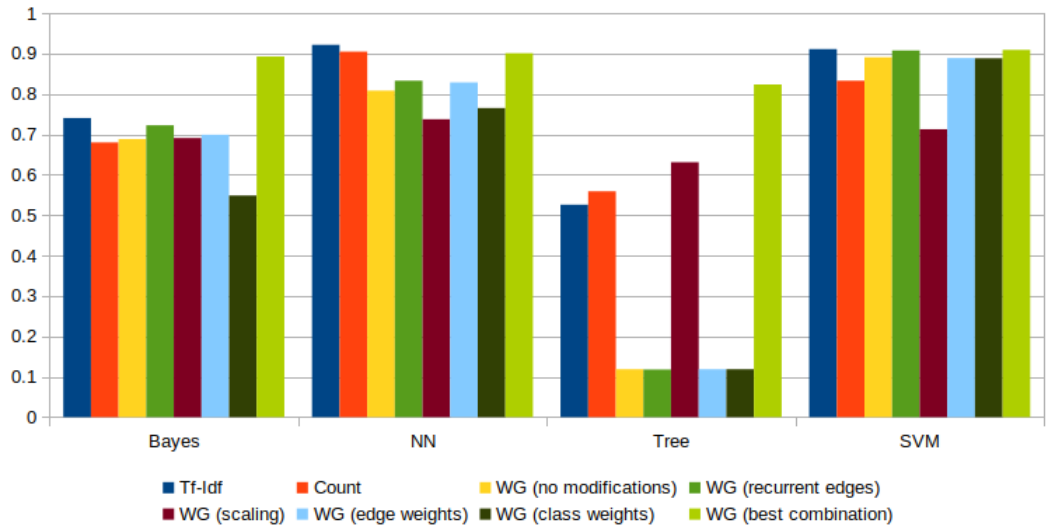

Figure 4.16: Classification accuracy of word-graph modifications for the balanced dataset

combination with the SVM classifier. On the other hand, the same modification greatly improved the decision tree model, outperforming the tf-idf and count vectorizers.

Tests on the imbalanced dataset, whose results are depicted in figure 4.17 show even more significant improvements. Most notably, the naïve Bayes classifier reached accuracy greater than 90%, while the decision tree accuracy was just below this threshold.

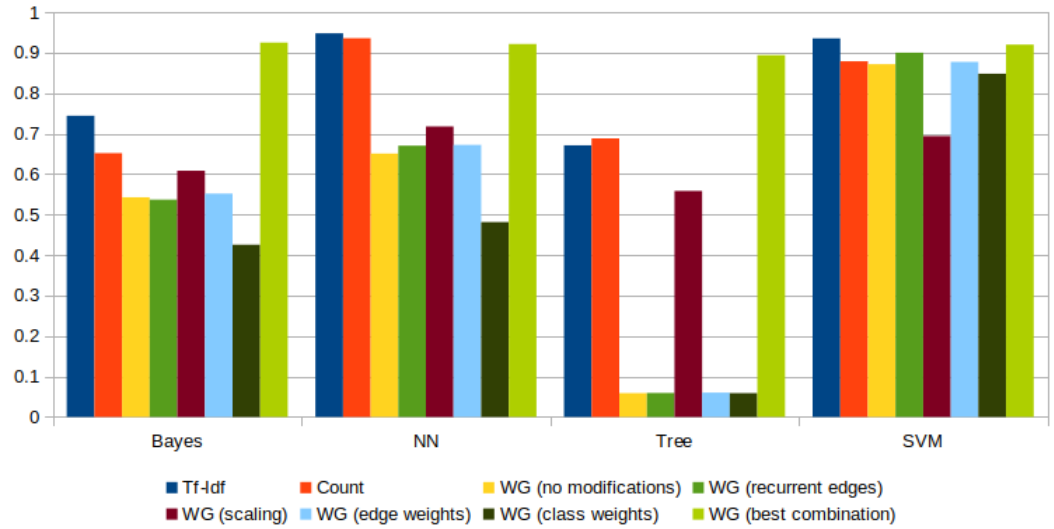

Figure 4.17: Classification accuracy of word-graph modifications for the imbalanced dataset

Although the decision tree classifier still provides the lowest performance compared to other models, it can be preferred in certain situations. As mentioned in section 2.1.3.2, one of the biggest advantages of this approach is interpretability. Prediction results can be explained in terms of how "typical" the document is for each author.

When it comes to scaling, in most cases the normalization variant provided better results,

the only exception being the naïve Bayes classifier. For the neural network model, the differences between the normalization and interval scaling options was minimal and the situation could be reversed if the experiments were repeated.

In general, training times of the modified models were longer. However, no clear patterns could be observed in the data displayed in figure 4.18. One interesting exception is the decision tree classifier. We can see that variants with low classification performance actually required the longest time to train.

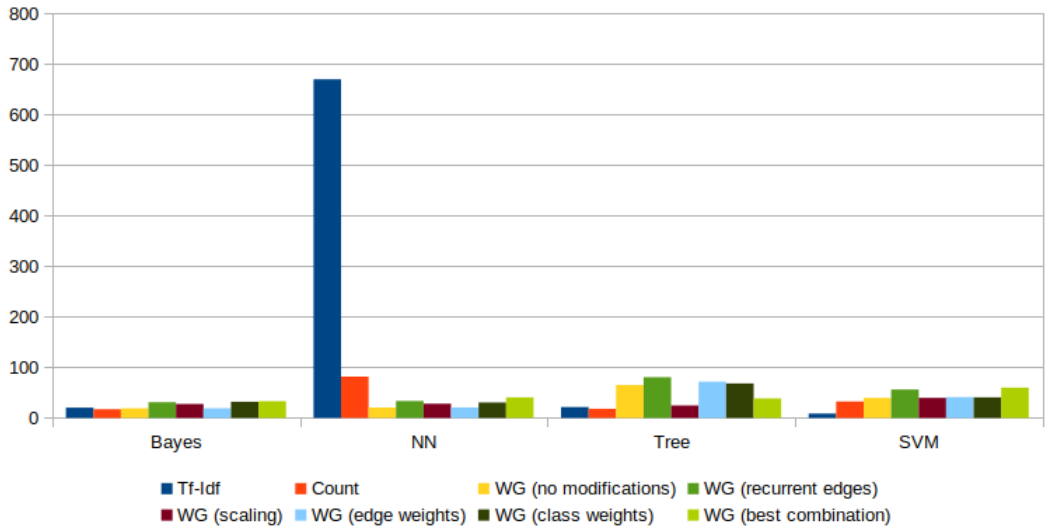

Figure 4.18: Training times of word-graph modifications for the imbalanced dataset

In conclusion, we find the modified word-graph vectorization method very useful in combination with the decision tree classifier. Although lacking slightly behind in accuracy, it provides the unique advantage of interpretability, which is required in many real-world applications.

5 Conclusion

As stated in introduction, the goal of the thesis was to evaluate multiple methods for solving a text classification problem - identifying an author of a text document written in Slovak. More specifically, we focused on vectorization - the process of creating a numerical representation of a text document. We compared established bag-of-words methods, the disadvantage of which is production of high-dimensional sparse vectors, with a novel word-graph approach.

Experiments performed were limited by available hardware resources. We reduced the task to a dataset containing 10 unique authors. Two variants were examined: balanced and imbalanced number of documents for each class. As shown, the bag-of-words methods provide better accuracy as the basic word-graph similarity method. Overall, we conclude, that linear SVM classifiers in combination with the tf-idf provide the best accuracy-training time ratio.

The situation changes after implementing a set of proposed modifications. For certain classification methods (decision tree and naïve Bayes classifiers) the word-graph approach provides the best accuracy. Decision trees provide an additional benefit of easy interpretation and the combination with the modified word-graph method can be useful in situations where model explainability is required or beneficial.

Given the limited resources and time, there are still many directions left to explore. Alternative similarity metrics for the word-graph model as well as other class weight coefficient variants should be further examined. As mentioned, the word-graph vectorization method was implemented "from scratch" as a proof of concept. Further implementation optimizations could potentially improve model's performance. We also completely ignored the word2vec/doc2vec approach to vectorization, which, based on literature review, also looks very promising.

Finally, the word-graph method can be further studied in context of other NLP tasks. As shown in section 2, text classification is closely related to problems such as clustering or regression. In all cases, the process of vectorization has to be applied and the word-graph method could provide a viable alternative.

References

1. AGGARWAL, Charu C. and ZHAI, Cheng Xiang. *Mining Text Data*. Springer Publishing Company, Inc., 2012. ISBN 978-1-4614-3223-4.
2. AGRAWAL, Rakesh and SRIKANT, Ramakrishnan. Fast Algorithms for Mining Association Rules in Large Databases. In: *Proceedings of the 20th International Conference on Very Large Data Bases*. San Francisco: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499. ISBN 1-55860-153-8.
3. AISOPOS, Fotis, TZANNETOS, Dimitrios, VIOLOS, John and VARVARIGOU, Theodora A. Using N-Gram Graphs for Sentiment Analysis: An Extended Study on Twitter. In: *2016 IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService)*. 2016, pp. 44–51. ISBN 978-1-5090-2252-6.
4. BIRD, Steven, KLEIN, Ewan and LOPER, Edward. *Natural Language Processing with Python*. 1st. O'Reilly Media, Inc., 2009. ISBN 0596516495, 978-0-5965-1649-9.
5. CASTILLO, Esteban, CERVANTES, Ofelia, VILARIÑO, Darnes, BÁEZ, David and SÁNCHEZ, Alfredo. UDLAP: Sentiment Analysis Using a Graph-Based Representation. In: *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*. Denver: Association for Computational Linguistics, 2015, pp. 556–560. ISBN 978-1-941643-40-2.
6. DENG, Li and YU, Dong. *Deep Learning: Methods and Applications* [online]. NOW Publishers, 2014 [visited on 2019-01-10]. Available from: `https://www.microsoft.com/en-us/research/publication/deep-learning-methods-and-applications/`.
7. GOLDBERG, Yoav and ELHADAD, Michael. splitSVM: Fast, Space-efficient, Non-heuristic, Polynomial Kernel Computation for NLP Applications. In: *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*. Columbus: Association for Computational Linguistics, 2008, pp. 237–240.
8. GRAVES, Alex, FERNÁNDEZ, Santiago, GOMEZ, Faustino and SCHMIDHUBER, Jürgen. Connectionist Temporal Classification: Labelling

Unsegmented Sequence Data with Recurrent Neural Networks. In: *Proceedings of the 23rd International Conference on Machine Learning*. Pittsburgh: ACM, 2006, pp. 369–376. ISBN 1-59593-383-2.

9. HAN, Jiawei, KAMBER, Micheline and PEI, Jian. *Data Mining: Concepts and techniques*. 3rd ed. San Francisco: Morgan Kaufmann Publishers Inc., 2011. ISBN 978-0-12-381479-1.
10. HARRIS, Zellig. Distributional structure. *Word*. 1954, vol. 10, no. 23, pp. 146–162. ISSN 0043-7956.
11. KLEMA, V. and LAUB, A. The singular value decomposition: Its computation and some applications. *IEEE Transactions on Automatic Control*. 1980, vol. 25, no. 2, pp. 164–176. ISSN 0018-9286.
12. LE, Quoc V. and MIKOLOV, Tomas. Distributed Representations of Sentences and Documents. *CoRR*. 2014, vol. abs/1405.4053.
13. LEVER, Jake, KRZYWINSKI, Martin and ALTMAN, Naomi. Principal component analysis. *Nature Methods*. 2017, vol. 14, pp. 641–642. ISSN 1548-7105.
14. LIN, Hsuan-Tien and LIN, Chih-Jen. A Study on Sigmoid Kernels for SVM and the Training of non-PSD Kernels by SMO-type Methods. *Neural Computation*. 2003. ISSN 0899-7667.
15. MCCULLOCH, Warren and PITTS, Walter. A Logical Calculus of Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*. 1943, vol. 5, pp. 127–147. ISSN 0092-8240.
16. PALMER, Shelly. *Data Science for the C-Suite*. New York: Digital Living Press, 2015.
17. RAMOS, Juan. Using TF-IDF to Determine Word Relevance in Document Queries. In: *Proceedings of the first instructional conference on machine learning*. 2003.
18. RASCHKA, Sebastian. *Python Machine Learning*. Packt Publishing, 2015. ISBN 978-1-7835-5513-0.
19. ROSENBLATT, Frank. The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. *Psychological Review*. 1958, pp. 65–386.

20. SARKAR, Dipanjan. *Text Analytics with Python: A Practical Real-World Approach to Gaining Actionable Insights from Your Data*. 1st edition. Berkely: Apress, 2016. ISBN 978-1-4842-2387-1.

21. VERT, Jean-Philippe, TSUDA, Koji and SCHÖLKOPF, Bernhard. A primer on kernel methods. In: *Kernel methods in computational biology 47*. 2004, pp. 35–70.

22. VIOLOS, John, TSERPES, Konstantinos, PSOMAKELIS, Evangelos, PSYCHAS, Konstantinos and VARVARIGOU, Theodora. Sentiment Analysis Using Word-Graphs. In: *Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics*. Nimes: ACM, 2016, 22:1–22:9. ISBN 978-1-4503-4056-4.

23. YANG, Ning, LI, Tianrui and SONG, Jing. Construction of Decision Trees based Entropy and Rough Sets under Tolerance Relation. *International Journal of Computational Intelligence Systems*. 2007. ISSN 1875-6891.

24. ZHANG, Ye et al. Neural Information Retrieval: A Literature Review. *CoRR*. 2016, vol. abs/1611.06792.

25. ŠVAŇA, Miloš and NĚMEC, Radek. Comparison of Linear SVM Algorithm Implementations in Python for Solving an Author Identification Problem. In: *Proceedings of the 21st International Conference on Information Technology for Practice*. 2018, pp. 153–160. ISBN 978-80-248-4196-0.

List of Abbreviations

ASCII - American Standard Code for Information Interchange

CPU - Central Processing Unit (processor)

CSV - Comma-Separated Values

FPGA - Field-Programmable Gate Array

GPU - Graphics Processing Unit

HTML - HyperText Markup Language

JSON - JavaScript Object Notation

KDD - Knowledge Discovery from Data

LDA - Linear Discriminant Analysis

LSI - Latent Semantic Indexing

MIT - Massachusetts Institute of Technology

ML - Machine Learning

NER - Named Entity Recognition

NLP - Natural Language Processing

NLTK - Natural Language ToolKit

NN - (artificial) Neural Network

NRSR - Národná Rada Slovenskej Republiky (National Parliament of Slovak Republic)

PCA - Principal Component Analysis

SVD - Singular Value Decomposition

SVM - Support Vector Machine

tf-idf - Term Frequency - Inverse Document Frequency

TPU - Tensor Processing Unit

UTF - Unicode Transformation Format

List of Figures

Declaration of Utilisation of Results from the Diploma Thesis

Herewith I declare that

- I am informed that Act No. 121/2000 Coll. – the Copyright Act, in particular, Section 35 – Utilisation of the Work as a Part of Civil and Religious Ceremonies, as a Part of School Performances and the Utilisation of a School Work – and Section 60 – School Work, fully applies to my diploma (bachelor) thesis;
- I take account of the VSB – Technical University of Ostrava (hereinafter as VSB-TUO) having the right to utilize the diploma (bachelor) thesis (under Section 35(3)) unprofitably and for own use;
- I agree that the diploma (bachelor) thesis shall be archived in the electronic form in VSB-TUO's Central Library. I agree that the bibliographic information about the diploma (bachelor) thesis shall be published in VSB-TUO's information system;
- It was agreed that, in case of VSB-TUO's interest, I shall enter into a license agreement with VSB-TUO, granting the authorization to utilize the work in the scope of Section 12(4) of the Copyright Act;
- It was agreed that I may utilize my work, the diploma (bachelor) thesis or provide a license to utilize it only with the consent of VSB-TUO, which is entitled, in such a case, to claim an adequate contribution from me to cover the cost expended by VSB-TUO for producing the work (up to its real amount).

Ostrava, dated 24. 4. 2019

Miloš Švaňa

Student's name and surname

List of Annexes